A KeYmaeraX Visualizer

Chris Yu (christoy@cs.cmu.edu)

1 Abstract

We present a system for visualizing the physical systems described by hybrid programs. Our system interprets a hybrid program as an execution graph whose structure mirrors the control flow of the program. We then enable stepwise execution of the discrete statements of the hybrid program, combined with continuous numerical integration of differential equations.

2 Introduction

While hybrid programs inherently describe physical systems, there is currently no way to visualize the scenario described by a hybrid program, hampering one's ability to gain intuition for the system they are trying to model. While it is certainly possible to reason about hybrid programs from a purely mathematical perspective, ultimately hybrid programs are meant to describe real-world scenarios, for which the spatial intuition possessed by a modeler would be quite helpful.

Furthermore, while preservation of invariants is of course the most important aspect of hybrid programs, the behavior of the controller within the confines of the invariant is also important, when it comes to real-world deployment of controllers; for instance, even if a controller never leaves some designated safe area, it might still perform some safe but undesirable actions within the safe area, such as rapid acceleration or sharp turns. These behaviors may not become readily apparent over the course of a proof, but would be immediately observable once the program is actually run.

We have therefore designed a system to enable visualization of hybrid programs. Our system accepts a hybrid program written in the KeYmaera X language as input, and constructs a physical system mirroring the hybrid program. Internally, the program represents the hybrid program as an execution graph, with each atomic statement serving as a node. Executions of the program then correspond to different traversals of the graph, with nondeterministic choices represented as branching paths. The user is then able to step through the program. When a differential equation system is reached, the system then evolves the state of the system using numerical integration, the duration of which is controlled by the user.

Our system also implements tracking of preconditions and postconditions, which are input as a part of the hybrid program. At the start, the user is allowed to set the initial value of each variable. At this stage, our system provides feedback as to which preconditions are satisfied, and by how much they are violated, if so. Throughout the course of execution, the system also displays the desired postconditions of the program, and continuously tracks which ones are satisfied, and the error of those that are not.

3 Program representation

Program state in our system is represented as a tuple of four elements:

1. The current node in the *execution graph*.
2. The *environment* containing all values of variables.
3. The list of preconditions.
4. The list of postconditions.

We will discuss the implementation of each of these elements in sequence.

3.1 Execution graph

Within our system, programs are structured as a sequence of nodes, each of which enclose some elementary unit of computation. For instance, if we consider the following program, which models the scenario of a controlled car following a non-controlled leader car while avoiding collisions:

```
{
  {
    { ?(velCtrl < velLead); accCtrl := A; }
    ++
    {
    ?(velCtrl >= velLead &
              (posLead - posCtrl) >
              -(velCtrl-velLead)*(velCtrl-velLead) / (2*(-B)));
              accCtrl := A;
    }
    ++
    {
    ?(velCtrl >= velLead &
              (posLead - posCtrl) <=
              -(velCtrl-velLead)*(velCtrl-velLead) / (2*(-B)));
              accCtrl := -B;
    }
  }
  {
    { posLead' = velLead, posCtrl' = velCtrl, velCtrl' = accCtrl &
      (posLead - posCtrl) >= -(velCtrl-velLead)*(velCtrl-velLead) / (2*(-B))}
    ++
    { posLead' = velLead, posCtrl' = velCtrl, velCtrl' = accCtrl &
      (posLead - posCtrl) <= -(velCtrl-velLead) * (velCtrl-velLead) / (2*(-B))}
  }
}*
```

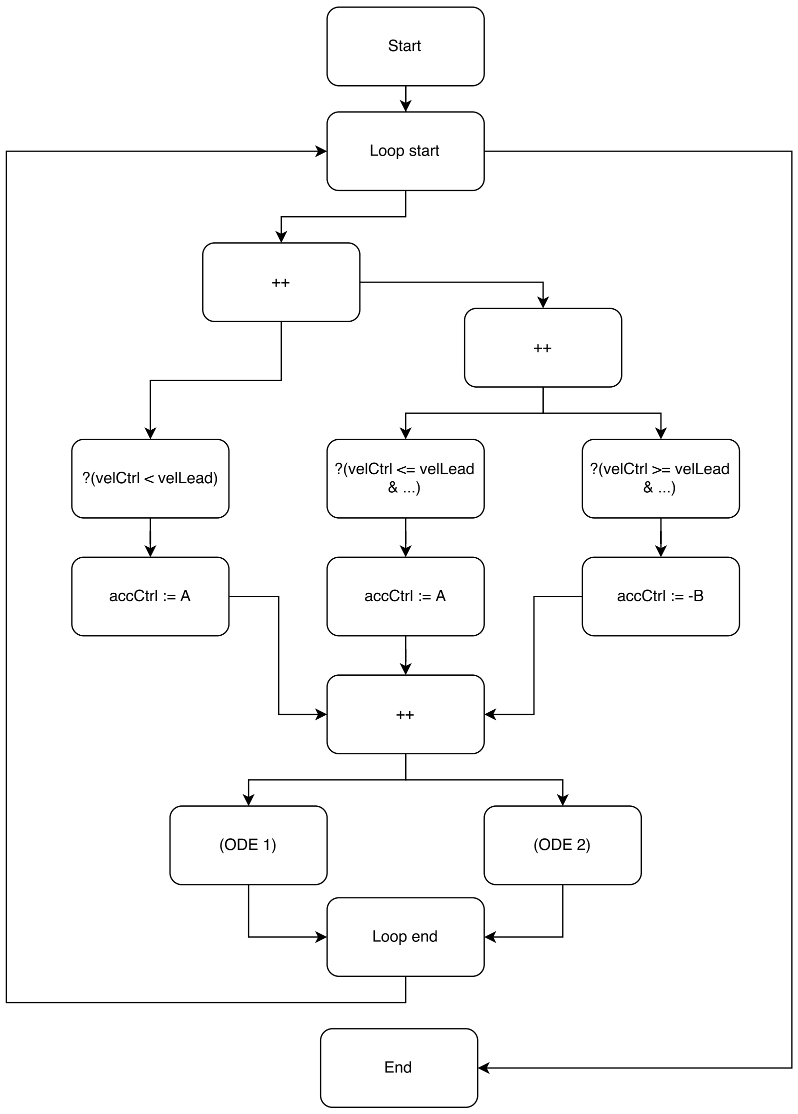


Figure 1: The graph of the hybrid program presented in section 3.

The resulting graph would have a structure resembling that of Figure 1. Any node with two arrows emerging from it is nondeterministic; program execution is allowed to follow either of the transitions. Note that loop nondeterminism is represented at the start node of the loop. Executing the program then reduces to simply performing a traversal of the graph. Each node is either an atomic operation, a differential equation system, or a no-op, and at each stage, there are at most two possibilities for where execution can proceed.

Constructing the execution graph is a fairly simple recursive procedure on the structure of a hybrid program. One slight subtlety is that, for each graph, we must define one **root node**, as well as a set of nodes that we will refer to as the **terminal nodes** of the graph. (Since execution graphs are not trees or even acyclic, these do not correspond to the leaves of any hypothetical tree.) We will address the possible cases below.

Atomic statements. These statements include assignments (both deterministic and non-deterministic) and tests. For these statements, we create a single node containing the atomic statement to be executed. The single node is both the root node and the only terminal node.

Differential equation systems. While these are not atomic statements in the same manner as the others, they have the same structure for purposes of the execution graph. We simply create a single node that contains the ODE system. The single node is again both the root node and the only terminal node.

Composition. The graph of a composition of two programs $\alpha;\beta$ consists of the graph G_α of α, followed by the graph G_β of β, where outgoing edges are added from all of the terminals of G_α, pointing to the root of G_β. The root node is the root of G_α, while the terminal nodes of this graph are the terminals of G_β.

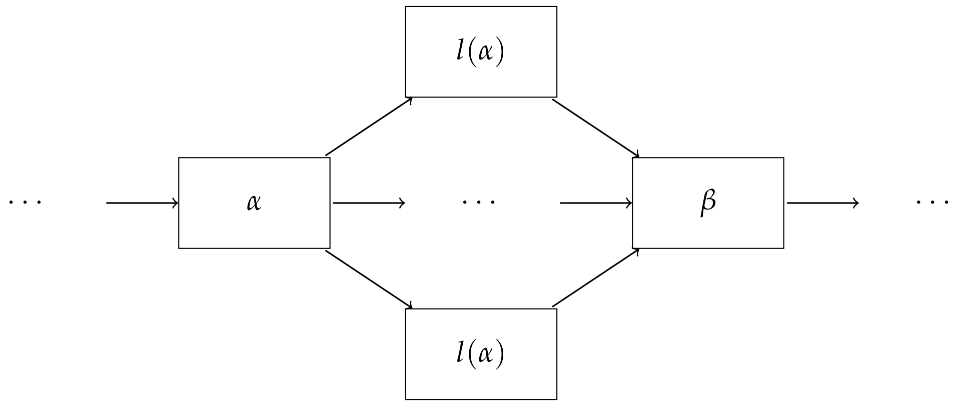


Figure 2: A simplified depiction of the execution graph for the composition $\alpha;\beta$. The terminal nodes of α are denoted as $t(\alpha)$.

Choice. To construct the graph of a choice between two programs $\alpha \cup \beta$, we first create a no-op node $n_\cup$ that represents the choice. We then construct the graphs G_α and G_β of α and β respectively, and add outgoing edges from $n_\cup$ to the roots of G_α and G_β. The terminal nodes of this graph consist of the union of the terminals from both G_α and G_β.

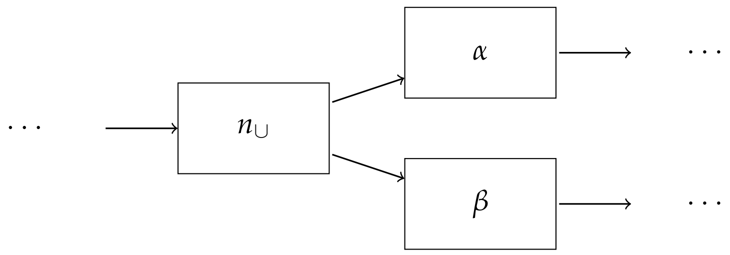


Figure 3: A simplified depiction of the execution graph for the composition $\alpha \cup \beta$.

Loop. In order to create the graph of a loop α^*, we first create the graph G_α of the inner program α. We then create two auxilliary nodes, n_{start} and n_{end}. n_{start} points to the root of G_α, while all of the terminal nodes of G_α point to n_{end}, which then points back to n_{start}. n_{start} then serves as both the root node and the sole terminal node of the whole graph. This enables a traversal to repeat the loop any number of times, including 0 times.

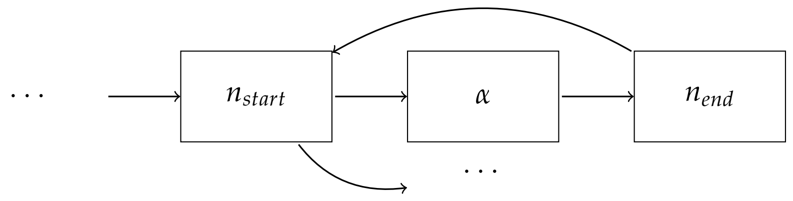


Figure 4: A simplified depiction of the execution graph for the loop α^*.

By recursively applying these procedures to an entire program, we create a complete execution graph for the entire program. We then add one additional node at the start, which points to the root of the entire graph, and we add one node at the end, which all terminal nodes of the graph point to; these simply serve to mark where the program starts, and when it has terminated.

3.2 Program environment

The environment of a program is simply a map from variable names to numerical values; it thus serves to store the values of all variables at all times. Assignments, tests, and ODEs are all executed with reference to the bindings in the environment. Any formula or term can be evaluated within the environment, by simply looking up the binding of any variable whenever it occurs within a term.

3.3 Preconditions and postconditions

Preconditions and postconditions, like all other formulas, are simply stored as KeYmaeraX formula objects. As just mentioned, these can be evaluated within any environment by looking up the bindings of variables. Beyond simply evaluating to true or false, however, we would also like to be able to compute how much error there is in an unsatisfied formula. Satisfied formulas should have 0 error, while unsatisfied formulas should have positive error.

We first compute a raw signed error, where negative error indicates that a constraint is satisfied with slack. This signed error can be computed recursively on the structure of formulas. We outline this computation below. Let $e(P)$ denote the error of a formula P.

- **Inequalities.** For an inequality $a \leq b$, we have $e(a \leq b) = a - b$. Note that for a strict inequality $a < b$, the signed error remains $a - b$, since the sole point of difference $a = b$ has measure 0, so the distance to the set is unchanged.
- **Equality.** For an equality $a = b$, we have $e(a = b) = |a - b|$.
- **Negation.** For $\neg P$, we have $e(\neg P) = -e(P)$. Intuitively, if P was false, then it had positive error, so the negation will have negative error and thus be satisfied.

- **Conjunction.** For $P \land Q$, we have $e(P \land Q) = \max(e(P), e(Q))$. Since both sides must be true, the total error is considered to be the max error on either side, with the conjunction achieving negative error only if both sides do.
- **Disjunction.** For $P \lor Q$, we have $e(P \lor Q) = \min(e(P), e(Q))$. The intuition is analogous to that of $P \land Q$, considering that in this case either P or Q could be true.
- **Implication.** We have $e(P \to Q) = \min(-e(P), e(Q)$. This leverages the equivalence $P \to Q \equiv \neg P \lor Q$.
- **Equivalence.** We have $e(P \leftrightarrow Q) = \min(\max(e(P), e(Q)), \max(-e(P), -e(Q)))$. This comes from $P \leftrightarrow Q \equiv (P \land Q) \lor (\neg P \land \neg Q)$.

We do not compute numerical errors for universal or existential quantifiers, since this would constitute a computational problem beyond the scope of the project. We also do not compute numerical errors for box or diamond modalities for the same reason, though we expect there is little reason to include these formulas in preconditions or postconditions.

After computing the raw signed error for an entire formula, we take the max with 0 in order to clamp it to positive values.

4 Program execution

Executing a KeYmaeraX program consists of performing a step-by-step traversal of the execution graph. Each step executes the operation stored in the current node (if any), and then transitions to one of its successor nodes. For deterministic nodes, this transition is automatically performed, since there is no choice to be made. For nondeterministic nodes, we prompt the user to make the choice. Since there are only ever at most two possible successor nodes, the user supplies the input by simply pressing one of two keys.

The actual semantics of the nodes follow those of hybrid programs. Assignments replace the current binding of a variable in the environment. A nondeterministic assignment does the same, but it prompts the user to supply the value that the variable will be replaced with.

Tests check the truth value of the supplied formula. If the test passes, no other action is performed. If the test fails, however, the user is informed, and program execution is temporarily blocked. The user then has the option of either restarting the program, or of forcing execution to continue anyway. The latter option causes execution to resume as if the test had not been performed, though of course the user should be aware that the remainder of the execution is not actually a valid execution of the program.

All discrete updates can be handled this way. As such, the only operation that requires additional attention is the ODE node, which encodes a continuous physical system. Once an ODE node is reached, the user is prompted to begin a physical simulation. The user is able to play and pause the physical simulation at will, and they have the option of ending the evolution of the system and proceeding to the next node at any time.

The exception is when the boundary of the evolution domain is reached. At this point, evolution of the system halts automatically, and is not permitted to continue. The user then has no choice but to end the evolution and proceed to the next node.

4.1 Numerical integration

Numerical integration of ODE systems is performed using a choice of one of two integrators. The simpler integrator can be described as either forward or symplectic Euler; the integrator that is actually manifested depends on the specific ODE system. In the most general case, the integrator is simply forward Euler, and uses the update rule

$$y_{n+1} = f(t_n, y_n)\Delta t$$

However, when the ODE system being integrated is second order and follows the general form $x' = f(t, v), v' = g(t, x)$, the integrator becomes symplectic Euler. This behavior is essentially achieved by updating the equations in sequence rather than simultaneously, and using the new velocity to update the position:

$$\begin{aligned} v_{n+1} &= v_n + g(t_n, x_n)\Delta t \\ x_{n+1} &= x_n + f(t_n, v_{n+1})\Delta t \end{aligned}$$

In terms of approximation error, both have a global error of $O(\Delta t)$. However, the advantage of symplectic Euler is that it is what is known as a symplectic integrator, which means that it roughly preserves volume in phase space. In practice, what this means is that a symplectic integrator approximately conserves energy in the system, which makes them suited for systems involving cyclical motion.

The other integrator that we provide is the Runge-Kutta fourth order method. This essentially constructs an approximation of the slope at the midpoint of the time interval using a weighted average of values from multiple Euler steps. In particular, we have

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

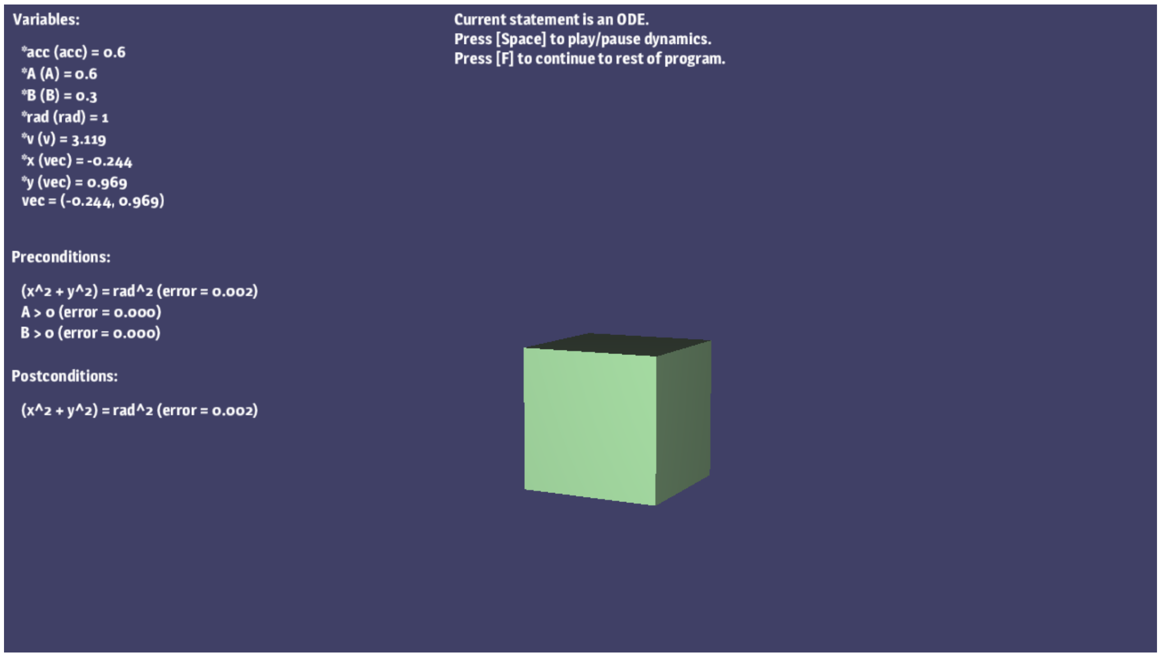
$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{aligned}$$

The advantage of this integrator is that it only commits a global error of $O(h^4)$. Thus, it will in general be more numerically accurate than forward Euler. However, it is not a symplectic integrator, so it will tend to exhibit drift in systems with cyclical motion; for this reason, such systems may still perform better when integrated using symplectic Euler.

By default, RK4 is used to integrate ODE systems, but the user is able to switch between the two options whenever desired.

4.2 Visualization

To display the simulation state, we have written a basic visualization system in OpenGL, which displays the program variables as small particles in 3D space. By default, all variables are interpreted as one-dimensional, and are thus confined to the x-axis. At the start, however, the user has the option of grouping variables together into vectors of up to 3 variables. These grouped variables will then be rendered as a single scene object with coordinates (v_1, v_2, v_3) corresponding to the three variables in order. Also at this stage, the user has the option of turning off rendering for any variables desired, for quantities such as time or velocity that do not correspond to physical objects.



```
Variables:

  *acc (acc) = 0.6
  *A (A) = 0.6
  *B (B) = 0.3
  *rad (rad) = 1
  *v (v) = 3.119
  *x (vec) = -0.244
  *y (vec) = 0.969
  vec = (-0.244, 0.969)

Preconditions:

  (x^2 + y^2) = rad^2 (error = 0.002)
  A > 0 (error = 0.000)
  B > 0 (error = 0.000)

Postconditions:

  (x^2 + y^2) = rad^2 (error = 0.002)
```

```
Current statement is an ODE.
Press [Space] to play/pause dynamics.
Press [F] to continue to rest of program.
```

Figure 5: A screenshot of our application. All variables have been hidden (indicated by an asterisk by their name) except for the spatial x and y coordinates, which have been grouped into a single vector called vec. The position of this vector is rendered as the small cube on screen.

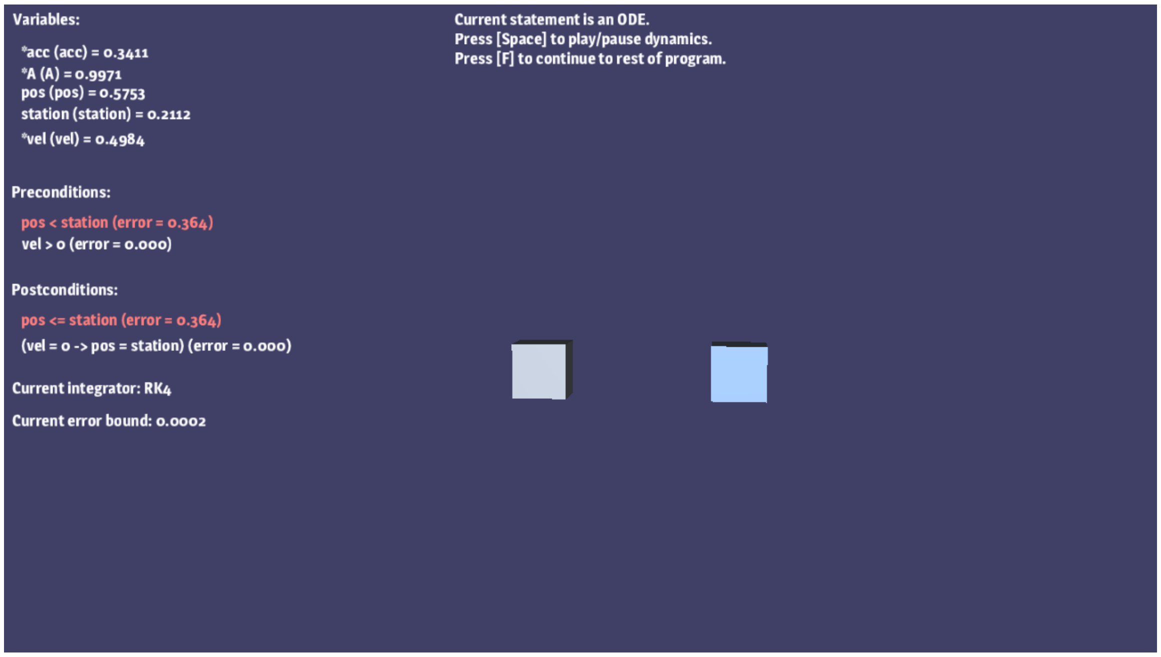
5 Constraint Checking

In addition to simply running the program, we track the values of preconditions and postconditions in real time. Since these are typically presented as a long conjunction of different formulas, we heuristically split up these conditions into these formulas based on conjunctions at the top level. It is then possible to check the truth value of each condition throughout the execution of the program, simply by evaluating the formula in the current state and environment.

Beyond simply checking the truth value, we also compute the error of each formula as previously described. We display the value at all times during simulation, and we shade each constraint with a shade of red corresponding to its current error.

Determining whether or not a formula is true or false is slightly more involved than simply evaluating it in the environment, due to the possibility of numerical errors. This can be mitigated by choosing a small timestep, but this of course comes at the cost of simulation speed.

We therefore take into account the bounds on global truncation error, which are known for the integrators we are considering; specifically, as discussed, forward Euler features a $O\Delta t)$ global truncation error, while Runge-Kutta (RK4) features $O(\Delta t^4)$ global truncation error. We therefore progressively increase our tolerance for when we consider an inequality or equality to be true, starting from 0 and increasing at some rate. Specifically, at each timestep, we add $O(\Delta t^2)$ to the current tolerance. This is somewhat more permissive than the full $O(\Delta t^4)$ bound claimed by RK4, but we still find it a reasonable tolerance for most systems.



```
Variables:
  *acc (acc) = 0.3411
  *A (A) = 0.9971
  pos (pos) = 0.5753
  station (station) = 0.2112
  *vel (vel) = 0.4984

Preconditions:
  pos < station (error = 0.364)
  vel > 0 (error = 0.000)

Postconditions:
  pos <= station (error = 0.364)
  (vel = 0 -> pos = station) (error = 0.000)

Current integrator: RK4

Current error bound: 0.0002
```

```
Current statement is an ODE.
Press [Space] to play/pause dynamics.
Press [F] to continue to rest of program.
```

Figure 6: A second screenshot of our application. The red coloration of the constraint $pos < station$ indicates that the constraint is currently unsatisfied. The degree of redness corresponds to the amount of error, which is also displayed numerically.

6 Discussion

Many of the implementation details for this system are fairly straightforward, owing to the highly structured nature of hybrid programs. Most computational tasks can be structured as a recursive computation on the structure of terms, formulas, and programs, making them quite easy to implement and debug.

One challenge, however, lies in the visualization of the program data. While we have drawn the program variables and vectors as small cubes on screen, the lack of any other details such as backgrounds or environments can make it difficult to judge their relative positions, particularly when it comes to depth. One might attempt to alleviate this by adding a floor, or axes, or other background elements; however, at this stage, we have not done so, since this only relates to the visual quality of the program and not its correct functionality.

All in all, we have accomplished most everything that we set out to do. We have built a visualization system that is able to execute hybrid programs, visualize their state in real time, and check the status of postconditions. There are a number of elements that we would have liked to have been able to add, if not for time, as we will discuss in the next section; however, these are not critical to the functionality of the system, and are more for convenience and utility.

7 Future work

A major way in which our system could be further improved would be in the handling of nondeterminism. At this stage, we simply ask the user to make all nondeterministic choices for the system, and if any of these choices results in a dead end in execution (e.g. a failed test), then the only recourse is to restart the program from the beginning. One thing that would be a useful addition would be an "undo stack" of sorts, where at any point, a user can choose to rewind execution to the last "point of nondeterminism", meaning a choice, a loop, or a nondeterministic assignment. If this is implemented as a true stack, then the user could rewind multiple times, allowing them to back up to a point that they might consider to have been the last safe point of execution. This would enable more efficient exploration of the possible execution trees of the program by enabling a user to rewind once a dead end is reached.

Another useful addition would be real-time tracking of not just postconditions, but internal program invariants as well, such as loop invariants. Often times, the difficult part of a proof is not the final postcondition, but the intermediate loop invariant that is used to arrive at the postcondition. We did not implement this tracking because we were not able to determine how to extract an invariant that was included in an `@invariant` statement in the program. However, implementing this would provide some more useful information to the user.

Tracking differential invariants could also be a useful task. This is not as straightforward, because these invariants tend to be entered directly into the proof assistant without first being annotated in the program text itself. However, one could consider implementing a method where the user could directly enter a desired invariant, which could then be added to the list of formulas for which to track errors.

Finally, the simulation flow could generally be streamlined; for instance, an auto function could be useful for skipping past deterministic nodes where there is no need for decisions, and therefore no interesting user input. As it stands, the step-by-step method of program execution is useful for conveying the most information about program state at each stage of the program, but if there is a long chain of assignments, for instance, one might only be interested in seeing the state at the end of the assignments, rather than after each one.

8 Conclusion

We have implemented a simple visualization system for programs written in the KeYmaera X model language. Our system is able to read programs, execute them, visualize the program state, and track the truth value and error of a list of postconditions in real time. We hope that this system will prove useful for future generations of aspiring proofmancers in CPS.