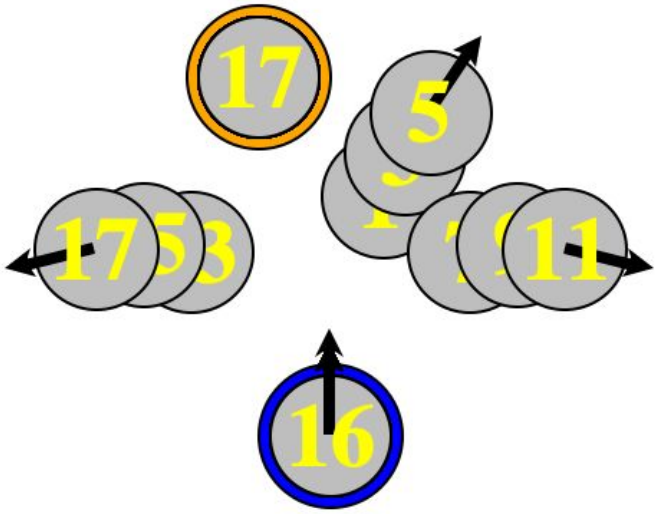
# Keymaera Evaluator for Reliable and Robust Cyber-physical Hybrid-program Engineering via Novel Graphics

Justin Kerr (jgkerr) and Jasmine Cheng (jacheng) (equal contribution)




**Abstract**

We present a visualization tool for hybrid programs that 1) parses annotated KeYmaera X models 2) creates a hybrid program tree and expands possible runs 3) displays a program run either through user choices or an automatic heuristic. It is general enough to visualize any number of 2D circular robots, so it can run any of the labs in the Logical Foundations of Cyber-Physical Systems course (15-424). This tool is useful for debugging incorrect models, and we show how it can illuminate problems with models from the course labs.

# Introduction

A common struggle for students first learning cyber-physical modelling is creating models that accurately represent the problem while also satisfying safety requirements.

In this project we create a visualization tool for students learning differential logic to help them understand their model behavior and identify possible pitfalls or errors. The visualizer helps students find unsafe scenarios to help them correct and constrain their models. The user is given

the option to visualize and select from available choices, or run the model automatically, where our system chooses dangerous paths of execution. This mode can expose edge case scenarios the user wouldn't have thought of.

We use our tool to visualize our models from all four labs in the LFCPS course. In addition, we produce some incorrect models and demonstrate how the visualizer can show shortcomings of these models.

In the "System Design" section we give a description of the various components of the visualizer, and in "Implementation" we go into detail about how we created them in code.

**Related Work**

In "A KeYmaeraX Visualizer'' from LFCPS Spring 2017, Chris Yu presents a visualization tool for stepwise execution of KeYmaeraX hybrid programs [3]. He represents the program using an execution graph constructed from atomic statements, differential equation systems, composition, choice and loop. He represents the program state via the node of the execution graph, the program environment with variable assignments, preconditions and postconditions. The visualizer itself prompts the users to step through the program and make all control decisions.

Our approach differs in a few important ways from Yu's project.

A. Our simulator uses the concept of "execution traces" to unroll individual loop iterations to try to pick antagonistic choices for the HP. This also allows the user to visualize possible execution paths and pick.
B. Our framework treats loops differently from Yu, unrolling them inductively step by step instead of embedding them inside a graph. Though this is less general (no nested loops allowed), it more naturally corresponds with proofs that most people do in this class (inductive loop invariants) and allows an iterative interactive simulation for the user.
C. Our framework includes automatic execution, where we pick actions that attempt to lead to a postcondition failure.

An important part of research in cyber-physical system verification is bridging the gap between abstract models and runnable implementations. This project relates to recent efforts to convert models into runnable code on robots. Prior work revolves around the idea of generating a verified runtime monitor for a given model, and checking during run time that the real system obeys the monitor[1,2] This work translates the monitor from differential dynamic logic into interval arithmetic, allowing it to be implemented in code. The models are implemented using CakeML, a verified implementation subset of StandardML. The resulting code can be compiled and run in simulations such as AirSim or on physical robots. Our project doesn't have as strong a theoretical foundation as this since its primary purpose is a debugging tool, but it operates in the same vein of bringing models into the real world.

# System design

**User Flow**

The target user group for our simulator is students of LFCPS who are learning modelling concepts. As such, we assume the user creates models using the KeYmaera X syntax, and would like to readily use these models without much modification for the simulator. The user flow is as follows

1. The user has a .kyx file with their model. The user annotates the model with a few indicators to suggest variables corresponding to robot properties. More details are outlined in the Implementation section.
2. The user runs the simulator with an input to the annotated model's filename.
3. The simulator invokes the parser and displays a program tree for the user to verify.
4. The visualization window opens. Users can choose "Manual" or "Auto" mode.
    a. Manual: On each loop iteration, the visualizer shows possible end states and the user chooses one to execute.
    b. Auto: The simulator makes choices via a heuristic on the post condition.
5. The simulator generates intermediate states, and displays an animation.
6. A text label shows the chosen program run

**Program tree structure**

The program tree is a representation of the hybrid program which follows a natural functional form. Each node corresponds roughly to some element of syntax, and stores children in the tree if relevant. For example, a Choice node stores its two children as subtrees. This recursive structure makes generating evaluation traces much easier since traversing the tree is equivalent to executing the program itself.

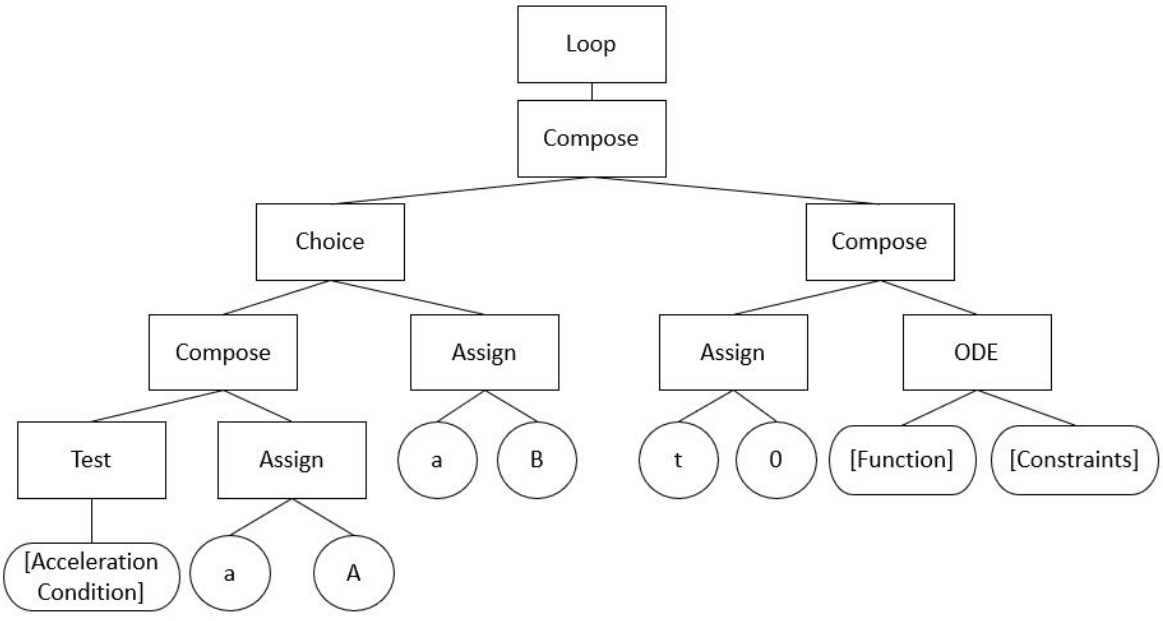



*Figure 1:* The parser creates a program tree by recursively finding the top-level operator.

**Loop treatment**

We allow the program to contain one top-level loop only. Otherwise, program trees would be ill-defined with ancestor loops because they would represent an infinite number of traces. Allowing them at top level lets us inductively simulate loop iterations, and though it is restrictive it comfortably allows for all labs used in this class, as well as a wide range of models of controllers.

**Program Traces**

An important concept in our system is a "program trace", which allows for the inductive simulation of loops, selection of choices, and handling of tests and constraints. Program traces represent a possible run of the program. Each trace represents one possible run of the program, and is a tuple of a choice list and the end state. The choice list contains choices in the order they are made. The end state contains the values of all variables after running the program with the given choices.

Given a start state and a program tree, we generate a list of traces using the recursive pattern of the program tree. Let $[a_1, a_2 \ldots a_n]$ be a list of choices for operators that cause the program to branch, i.e. Choice or ODE. Let T be a function that encodes the program tree, $T : choiceList \times startState \rightarrow endState$ and *traces* the list of traces. Then given a start state $s_i$,

$$traces = \{([a_1 \ldots a_n],\ s_f) \mid T([a_1 \ldots a_n],\ s_i) = s_f\}$$

Generating program traces at the beginning of each run provides several key advantages. First, it allows a natural way to select choices. By computing all possible end states, either the user or the auto heuristic can make a selection based on conditions of the end states. Once the trace is selected, it can easily be simulated by traversing the program tree via the choices list and updating the state accordingly. Additionally, traces provide a more general framework than making choices during run-time, which can be extendable to further applications that require knowing possible program states ahead of time.

Moreover, pre-computing all branches ensures that branches which would fail a Test operator during their run are pruned. If choices were made at run-time, then the user or auto-chooser may make a series of choices that end up with the program failing a Test, invalidating the entire run. Thus, we only keep traces that are guaranteed to execute.

**ODE run time**

We chose to treat ODE similarly to choices, splitting traces along the possible execution times of an ODE. We chose arbitrary time increments to try evaluating the ODE for and return as intermediate states in the trace which correspond to no time, a short interval, medium-length interval and long interval. We also check for domain constraint satisfaction during ODE evaluation, stopping the execution as soon as the constraint is violated.

**Antagonistic automatic trace selection**

Our system attempts to pick antagonistic traces which bring the system as close as possible to breaking the postconditions. We accomplish this by defining a heuristic for "distance" to the postcondition, and then picking the trace which results in the lowest distance to the postcondition at each loop iteration. This method is greedy, yet has a compelling ability to break programs at the level of complexity of the course labs.

We define the notion of distance recursively for each formula operator and conjunction. We first make a simplifying assumption that equality always has infinite distance, since under floating point arithmetic it is functionally impossible to reach strict equality. So, we only consider inequalities in our distance heuristic. Then, the distance of an inequality is the absolute value of the difference between the two terms. To combine distances across operators, "logical and" takes the minimum of each conjunct, and "logical or" takes the maximum, since "and" requires both to be satisfied while "or" only requires one.

**Tree Visualization in Manual Mode**

In manual mode, we visually display robot positions of the possible outcomes from the traces for the user to choose from. Rather than showing a list of choices, the visualized states give users a better idea of the possible results. The list of choices itself is not immediately useful since the user would need to mentally cross-reference with the model to figure out what the program will do.

# Implementation

**Software overview**

The system is implemented in Python, using minimal external packages. We use basic string parsing functions for the parser, and TKinter, a simple graphics framework, for visualization. Everything else is implemented from scratch. We present a modular design that supports the pipeline from raw text to graphical visualization.

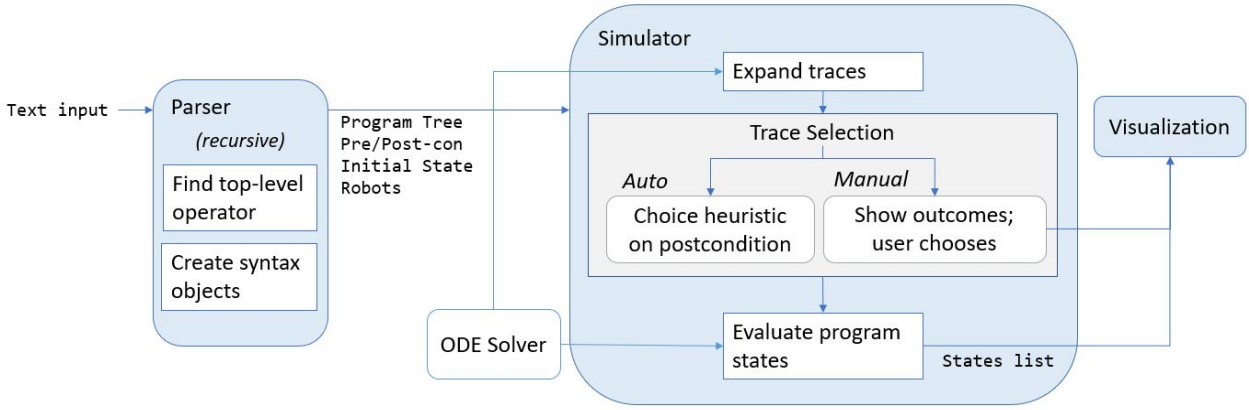



*Figure 2*: The system architecture is composed of a parser, a simulator engine and a visualizer.

- Parser: The parser is responsible for evaluating the string input model written using KeYmaera X syntax. It recursively constructs a syntax tree by finding the top level operator, creating an object to represent this operator and evaluating the children on each side of the operator.
- Program tree: This data structure holds the recursive structure of the program, corresponding loosely to a syntax tree of the program.
- Program simulator: The simulator will run the hybrid program and make control choices during the run of the program.

- ODE Solver: The solver is used by the simulator to integrate state variables via ODEs.
- Visualizer: The visualizer will create an animated graphical representation of the hybrid program using the state from the program simulator.

**Parsing and Program Tree**

*Robot annotations and start state*

The user is required to specify the initial values of free variables upon program entry, or in other words any variable that if left unspecified would result in no valid value for an expression at some point in the program. We choose manual annotation since it is the most useful for users to debug their models by trying exact initial conditions.

In addition, certain variable names can be annotated as variables to visualize. The visualizer supports naming any number of robots, where each robot has an x,y coordinate, radius, and dx,dy direction vector. These are automatically drawn by the visualizer corresponding to the annotations.

*Program parsing*

The parsing component of the system is responsible for taking the raw KX syntax and translating it into a program tree.

We require the inputted model to be of shape (*pre-condition*) -> [*program*](*post-condition*). After separating the pre and post-condition, we parse the program inside the box. The parser works recursively by finding the top-level syntax operator of each string, then splitting the string into its smaller parts and recursively parsing each substring. We respect operator precedence and binding, namely from strong to weak: ?, :=, ;, ++. The parser produces a program tree as its end result.

Our parser differs from Keymeara X in a few ways for ease of implementation.

- Quantifiers (for all, there exists) are not allowed.
- We support only box programs, not diamond programs
- Semicolons always have to be between phrases, no dangling semicolons inside brackets. For example, {a};{b} not {a;}{b}.
- No nested loops are allowed.
- Doesn't support "!=", instead just put "!(x=1)" for example.

*Program tree*

The program tree contains nodes that are HP derived classes. HP is a template class that represents a hybrid program operation. The derived classes are Choice, Loop, Assign, Compose, Test, and ODE. Each derived class inherits from HP and implements an "eval" and "expand" function, which will be described in more detail in the Program Traces section.

Nodes contain children which can be other HPs, a 'Form' object, 'Term' object or string. The Form class represents a formula and implements an eval function that returns the truth value of the formula given a state. The Term object represents an expression, and implements an eval function that evaluates the expression, returning its real number value given a state.

Below is a table showing the HP derived classes and the children types of each class.

*Table 1:* HP derived classes and their children types.

| | |
|---|---|
| Choice | lhs: HP<br>rhs: HP |
| Loop | arg: HP |
| Assign | x: string (representing a state variable)<br>e: Term |
| Compose | lhs: HP<br>rhs: HP |
| Test | arg: Form |
| ODE | arg: string (representing ODE and constraint) |

*Formula and term evaluation*

Formulas and terms in our system are not represented by syntax trees, but simply an executable Python String. So, for example the formula "x>2 & y>0" is represented by "x>2 and y>0", so that to query the truth value we can call the builtin 'eval' function in Python. Likewise, to evaluate terms we simply swap out the relevant syntax (such as "**" for "^") and evaluate the phrase for its numerical value.

**Program Traces and Trace Selection**

We use the program tree to generate a list of traces by recursively calling the "expand" function on the nodes of the tree. Each trace represents one possible run of the program, and is a tuple of a choice list and the end state. The choice list contains choices in the order they are made. The end state contains the values of all variables after running the program with the given choices.

In each of the sections below we show a brief example of a program, "T", and how the corresponding traces calculation proceeds. Function `traces(T, s)` corresponds to expanding traces for program tree T given start state s. States are shown as a dictionary `{var1: value, var2: value}`.

*Assign*

We simply update the state, and do not need to prepend anything to the choices lists since no choices are being made.

```
T = {x := 0}
traces(T, s) = [([ ], {x: 0})]
```

*Star assign*

The expression "x:=*" represents a non-deterministic assignment in Keymaera X, meaning that after the assignment x can take any real value. Star assignment is an important part of many of the later

labs. To support this option, we intuitively thought of star assignment as a sort of continuous choice, and discretized the choice to provide a branching factor with each option in a separate trace. This behavior lets the user view outcomes corresponding to a representative sample of the star assignment, as well as allows the simulator to prune out failed tests naturally as in normal choices.

We require that the user specify the range and resolution to sample from, which covers most practical usages of the operator where the model chooses a value for a control input.

```
T = {x:=*(-1,1,3)}
traces(T,s) = [
      ([*(-1)], {x: -1}),
      ([*(0)], {x: 0}),
      ([*(1)], {x: 1})  ]
```

*Choice*

Given a current state s, we expand the traces for running the left branch starting at s, and the right branch starting at s. Then, for each trace returned by the left branch, we prepend the choices list with "L", and similarly we prepend "R" to each trace in the right branch. We return the concatenation of these two trace lists.

```
T = {x:=0} ++ {x:=1}
traces(T.left, s) = [([ ], {x: 0})]
traces(T.right, s) = [([ ],  {x: 1})]
traces(T, s) = [
      ([L], {x : 0}),
      ([R], {x : 1})]
```

*Compose*

Given a current state s, we expand the left child of the compose and get all traces from running the left hand side. Then, for each of the traces from the left child, we get the traces from running the right child starting at the left child's end state. We append the choices made by the right child to the choices made by the left child.

```
T = {{x:=0} ++ {x:=1}}; {{x:=x+1} ++ {x:=x-1}}
traces(T.left, s) =  [([L], {x : 0}), ([R], {x : 1})]
traces(T, s) =  [
      ([L, L], {x : 1}),
      ([L, R], {x : -1}),
      ([R, L], {x : 2}),
      ([R, L], {x : 0})]
```

*Test*

We evaluate the test condition. If the test succeeds, we return a list containing a single trace with the end state and no choice choices. If the test fails, we return an empty list since there are no possible runs.

```
T = {?x=0}
If s = 0
      trace(T, s) = [([ ], {x : 0})]
Else
      trace(T, s) = [ ]
```

*ODE*

Since it's impossible to evaluate the ODE for all run times, we instead pick a few time periods to run the ODE for. Thus, we can treat the ODEs like a multi-"Choice". Specifically, we evaluate the ODE for time 0, 0.1, 1 and 10. We append the time run to the choice list, taking the constraints into account. If we hit a constraint before reaching the run time, we return the time it stops at.

```
T = {x' = 1 & x < 6}
s = {x : 0}
trace(T, s) = [
      ([0], {x: 0}),
      ([0.1], {x: 0.1}),
      ([1], {x: 1}),
      ([5.99999], {x:5.9999})
]
```

**Simulation and Visualization**

*State*

We represent state as a tuple of values. We use a dictionary to map indices of tuple items to variable names. Each value is either a real number or nan (not a number) if undefined.

Our original implementation used a dictionary mapping variable names to values, but became too slow for ODE integration. The tuple representation allows for fast arithmetic operations on states such as adding and scaling.

*Simulator*

The simulator is the top level program that is executed by the user. Given a string input in the KeYmaera X syntax (extended with some of our Simulator specific modifiers), it calls the parser to generate a program tree, and then expands the tree to get traces. It opens up a simple UI through which a user can either choose "manual" or "auto".

The behavior of "manual" depends on whether the hybrid program has loops. If there are no loops, the simulator generates a list of program traces. Via the visualizer, the user chooses which trace to run, and the trace is executed. If the hybrid program contains an outer loop, then at the start of each loop, the simulator generates a list of traces for one loop iteration, from which the user chooses.

For auto, the simulator generates program traces and chooses which trace to execute via our post condition heuristic. Similarly to manual, if the hybrid program contains an outer loop, the simulator will generate program traces at the beginning of each loop for the program inside of the loop and choose which trace to run.

To execute a trace, we call “evaluate” on the program tree recursively, using the list of choices from the trace, and accumulating a list of states. For example, a Choice node will consume the first element from the choices list which should be either “L” or “R”, then call evaluate on the corresponding child with the remaining elements of the choices list. For an ODE node, it will evaluate the ODE for the time given by the first element in choices, and add a state to the state list for each time step we want to visualize. For Assign, no choices are consumed and a single state is added to the states list with the updated variable. Similarly, Compose does not consume a choice and calls its left child, then its right child with the remaining choices, and concatenates the states lists of both children.

*ODE integration*

We use the Runge-Kutta method for integrating ODEs which computes a weighted average of approximated slopes at multiple forward time steps. In general, this method yields smaller numerical errors than the Forward Euler method.

*Visualizer*

We implemented a simple 2D visualizer user Tkinter, a common light-weight 2D graphics library in python. Robots are represented by their x-y position and a given radius, and each is given a color. They are displayed on a fixed-size canvas where we use a constant ratio between pixels/unit and the center of the canvas is the origin (0,0).

The Visualizer steps through the list of states provided by the Simulator, updating the robot positions and pausing by the time step interval between updates. We display information about the robot states, as well as which branches of the hybrid program it is taking.

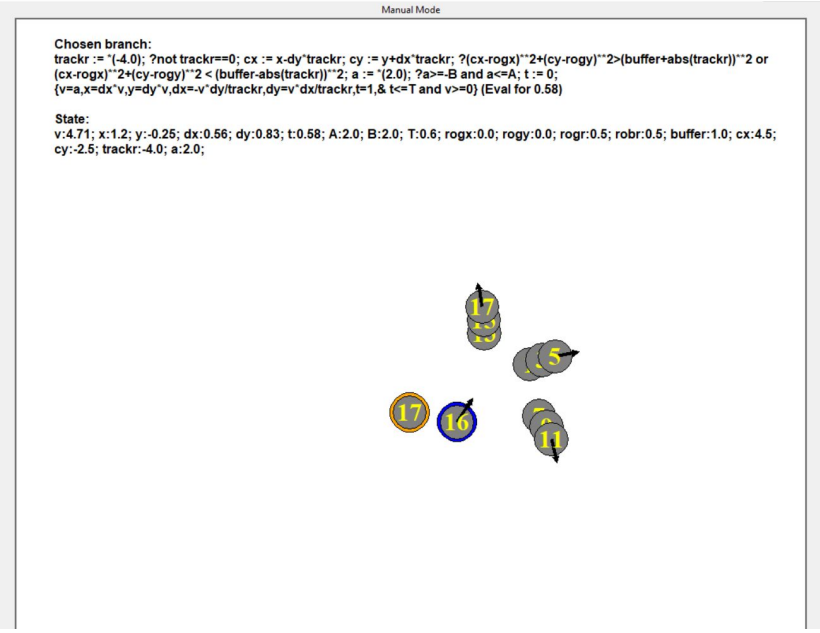



*Figure 3:* Screenshot of the Visualizer's user interface in manual mode.

# Results

In this section we demonstrate the capabilities of our simulator on the course labs. We show how it can be used to spot a variety of mistakes in models, as well as showcase the automatic model-breaking capabilities it has.

### Modelling errors

One of the most useful applications of this simulator is to visually identify modelling errors that students make: for example incorrectly setting the equations of motion for a differential drive robot will immediately produce visibly incorrect behavior.

The most difficult lab to model is lab 4, where the direction of the robot needs to evolve correctly along with its position. Luckily, this is visually a very easy condition to check, so our visualizer can very easily verify it. Below are selected modelling errors which produce obvious visual errors, along with the correct version.

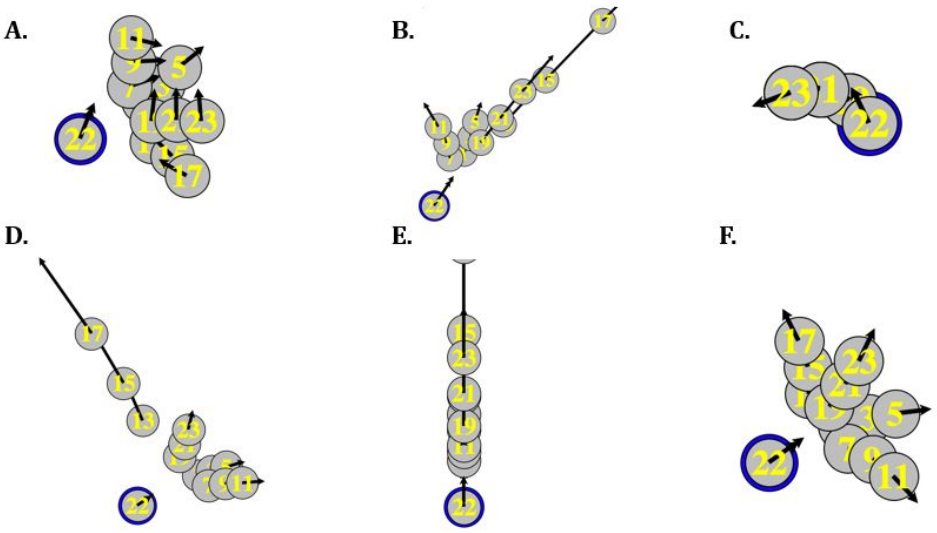



*Figure 4*: Trace generation visually indicates incorrect models. A. swapped x' and y', B. missing negative sign, C. forgetting to scale by track radius, D. typo on dx/dy, E. swapped dx/dy, F. correct version

**Controller errors**

Controller errors are typically much more subtle, but even our simulator is useful in checking controllers by interacting them in manual mode as a sandbox.

The two images below are taken from lab 4, where the blue, controlled robot should drive freely in the plane while avoiding the orange static obstacle. A correct controller permits steering commands where the robot's trajectory does not intersect the obstacle, which can be represented by requiring the obstacle is either inside or outside the robot's circular track. In this first example, the controller only includes the "outside" case, so the allowed trajectories are too conservative. Only two out of the four available steering commands are chosen, since the other options would cause the obstacle to be inside the track.

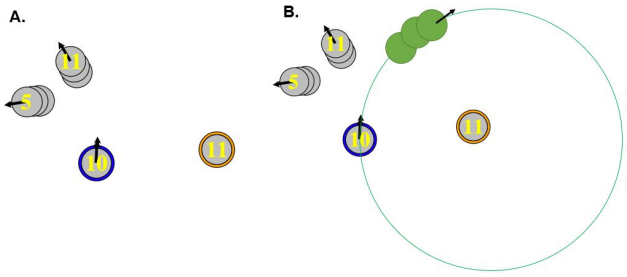



*Figure 5:* The visualizer shows over-conservative control mistakes. A. Options available to the controller. B. The green circles are options that should exist if the controller was correct, where it allows the obstacle to be inside the robot's track.

The following image shows the “inside” case, where the controller is so conservative that no choices are possible since it does not allow tracks where the obstacle is outside.

*Figure 6*: an even more restrictive controller has no available control inputs.

Another example of a broken controller is in lab 3, where the robot is constrained on a single circular track and should avoid an obstacle on the track. The image below shows a mistake in the controller where the robot doesn’t consider the distance it covers in the next timestep, leading to a collision.

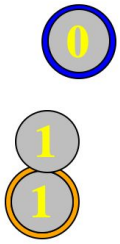



*Figure 7*: Our visualizer catches a broken acceleration controller in lab 3

**Antagonistic trace selection**

Our heuristic for choosing traces in auto mode can lead to automatically breaking models, such as in lab 3, shown below. The robot’s controller is unsafe and thus results in a collision.

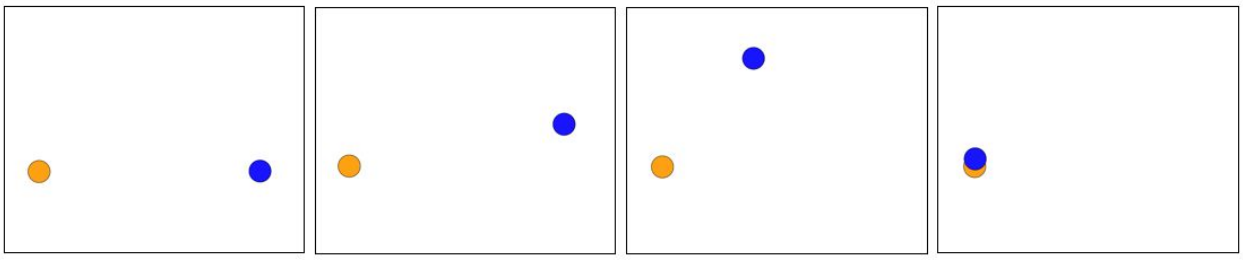


*Figure 8:* Time lapse of automatic execution resulting in a collision on a broken lab 3

Even with safe controllers, the automatic heuristic selection results in interesting behavior where the simulation makes choices that result in as dangerous a trajectory as possible. In lab 4, you can see the robot skirt slowly around the obstacle, just barely not colliding with it. This behavior is shown in the figure below.

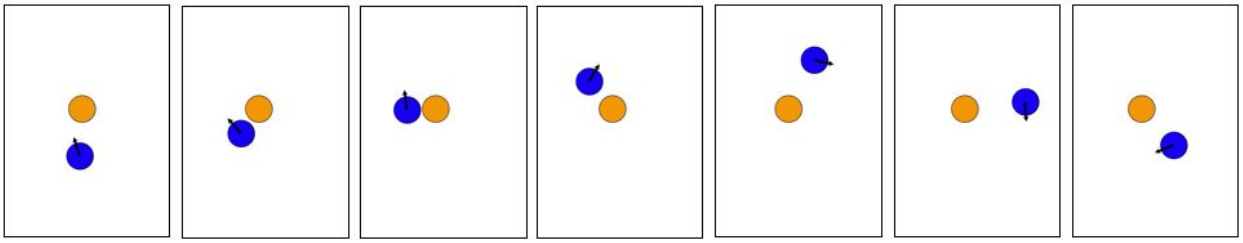


*Figure 9:* Time lapse of automatic execution resulting in risky trajectory for correct controller

Finally, below is an example of a broken controller which leads to the robot repeatedly crashing into the obstacle under automatic selection.

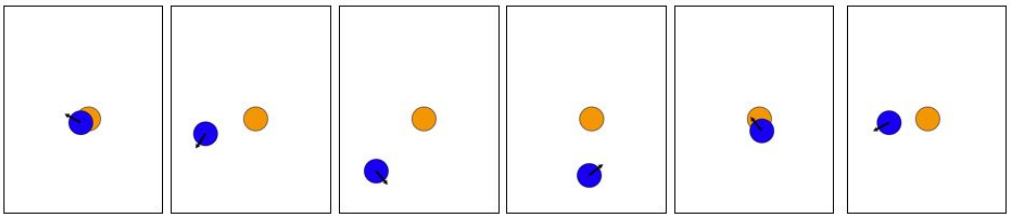


*Figure 10*: Faulty lab 4 controller results in collision

# Discussion and future work

All in all, we accomplished what we set out to do by making a useful tool for debugging labs in this course by visualizing their execution. We made several simplifications, especially in program tree parsing and formula evaluation, but this was an acceptable choice since it gave us the space needed to implement the rest of the system. Though a bit rough at the edges, with another semester of effort this could very well be turned into a streamlined experience for the user.

A key challenge was designing the program representation and execution system. The idea of generating program traces was not immediately apparent, since we were concerned by the large number of possible branches due to choice, ODE run times, and star assigns. We would need to evaluate each branch to generate the end states while creating state "frames" at fine time-steps for the visualizer. This led to the 2-pass design where we first evaluate each branch to generate its end state, and then for the chosen trace, re-evaluate and interpolate ODEs to create intermediate states for visualization.

Furthermore, we spent a considerable amount of time developing the parser and program evaluator before even working on simulation and visualization. This made us appreciate the meticulous work for designing proovers such as KeYmaera X, and pushed our understanding of hybrid program syntax rules.

**Automatic trace selection**

The heuristic we chose for trace selection was sufficient in some simple cases, but in more complicated ones it can fail because of its greedy nature and inability to consider intermediate states. There are two directions one could pursue to address these issues:

*Multi-level game tree search*

Similar to how game AI's function, the concept of traces is easily extendable into a graph search on possible choices multiple loop iterations into the future. Using our heuristic, a guided search similar to A* could be used to prioritize search along promising traces, with cost between states being the time between them. In our case, we don't care about the least cost path, but rather that *any* path to a failing state exists, so inadmissibility of the heuristic is unimportant.

*Differential heuristic*

Because our system only considers end states of trajectories, it is unable to detect when an ODE passes through a postcondition-breaking state. To remedy this, an approach inspired by the dI rule could be used: halt execution of the ODE when the derivative of the heuristic value switches from negative to positive. Intuitively, this represents a local minimum in the heuristic, which is a good place to stop ODE execution in a greedy sense. To preserve other options to support game tree search, this local minimum ODE time could be added as a separate trace, preserving other choices of runtime.

**User experience**

Additional features could enhance the visualization experience and allow for greater flexibility. Currently, the viewing canvas frame is fixed; canvas panning or zooming would allow users to view robots that travel longer distances. We only support circular robots; support other shapes like

rectangles and triangles can be added. In addition, more complicated visualization of traces, interactive syntax debugging, and more fine-grained options would greatly improve the usability.

**Mitigating numerical error**

One of the central challenges in any simulator will be integrator and floating point operation error, which unavoidably will be present in any computer simulation. To mitigate this in select scenarios, the simulator could use symbolic integration to solve ODE's which are simple enough to generate closed form solutions. Then, with symbolic support of formula evaluation the exact value of the function could be used in post-conditions and domain constraints. This is especially useful for post-conditions that depend on equality operators. While our current implementation is able to evaluate post-conditions after a program run, error from ODE integration makes satisfying equality conditions almost impossible. Alternatively, a tolerance on equality conditions may be added.

**Modelling event-driven controllers**

The most difficult problem we ran into was accurately simulating overlapping domain constraints as in event-driven controllers. Numeric error is impossible to get rid of, so checking for equality between floats on the boundary of constraints is not feasible. In addition, since the step size of the ODE integration is discrete, it's likely the boundary will never be crossed and the simulator will be stuck. We couldn't think of a good way to solve this problem, but luckily most models used in this class are time-controlled.

# References

[1] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Andrew Sogokon, and André Platzer. A Formal Safety Net for Waypoint-Following in Ground Robots. IEEE Robotics Automation Letters. 4(3), IEEE, 2019.

[2] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: Verified Controller Executables from Verified Cyber-Physical System Models. Programming Language Design and Implementation - 39th ACM SIGPLAN Conference, PLDI 2018, Philadelphia, PA, June 18-22, 2018, ACM, 2018.

[3] https://lfcps.org/course/fcps17/projects/christoy.pdf