Annika Peterson
Charlie Swanson
15-424 Term Paper

Modeling Wall•E and Eve's Love

Describing the System:

We studied a system based on the movie Wall•E. During the movie, there is a scene where Wall•E and Eve celebrate their love by dancing through space. During the dance, they twirl around each other in a spiral. During this spiral, they make sure that they keep up with one another (meaning they are never too far apart from each other), and more importantly ensure that they never crash into each other. Wall•E is able to fly because of a fire extinguisher, while Eve has the ability to fly without help. Due to the extinguisher's space-travel limitations, Wall•E's controller has to choose discretely to accelerate or brake, and could have a hard time keeping up while Eve is traveling at a constant velocity. We created a control for Wall•E to make sure Wall•E never hits Eve during their dance.

We model the system as Wall•E and Eve moving along a spiral centered at the origin. The spiral is a circle in the x-y plane that moves forward with some velocity in the z direction. We have two models, one which is simply modeling both traveling at a constant velocity and staying on opposite halves of the circle, the second models Wall•E as he attempts to keep up with Eve who is traveling at a constant velocity. The first model (simple-test.key) uses the regular x-y plane and z direction. To ease in proving the second model, we modeled the motion using polar coordinates. The circle is of radius cute_r, which Wall•E and Eve should never leave. When they are on the circle, we ensure that the sum of Wall•E's and Eve's radii is less than cute_r so that they both can fit on the circle of motion without hitting each other.

There is one property that is the core of the system for safety and for liveness. The safety property is that Wall•E and Eve never crash into each other. We also prove in our simple test that they are always on opposite sides on the circle from each other, which implies that they don't crash.

Methods to Analyze the System and Key Files:

We analyzed our system in two different ways, both of which are described below. The simple-test.key is a simplified model of the system where Wall•E and Eve are both uncontrolled, but move along the spiral. The second found in follow-eve-polar-test.key is a controller for Wall•E that allows him to break and accelerate to follow Eve, who is moving with constant velocity, around the circle without hitting her.

simple-test.key

To model the constant velocity system, we begin with the assumption that the distance between Wall•E and Eve is (2*cute_r)^2. This is equivalent to them being on opposite sides of the circle. We then show that if they stay at the same speed, they will always be on opposite sides of the circle. Since they are always (2*cute_r)^2 apart from each other, this implies that they never hit each other as cute_r > 0. Using KeYmaera, we proved this property by having the following important differential invariants.

`x_w = cute_r*dy_w` (Equation 1)

`y_w = -cute_r*dx_w` (Equation 2)

`x_e = cute_r*dy_e` (Equation 3)

`y_e = -cute_r*dx_e` (Equation 4)

`dx_w^2 + dy_w^2 = 1` (Equation 5)

`dx_e^2 + dy_e^2 = 1` (Equation 6)

`(x_w - x_e)^2 + (y_w - y_e)^2 = (2*cute_r)^2` (Equation 7)

Equations 1 through 4 help to ensure that the direction of motion of Wall•E `(dx_w, dy_w)`, and Eve `(dx_e, dy_e)` correspond to their x and y position on the circle. Equation 5 and 6 describe the vectors of motion as they move around the circle. The most important invariant is equation 7, which is our safety property.

follow-eve-polar-test.key

follow-eye-polar-test.key is designed to model the situation where Eve moves with constant velocity and Wall•E tries to not run into her. Note this also means that Wall•E has to ensure Eve never runs into him (which is possible since Eve is moving at constant velocity). Our controller for Wall•E has non-deterministic acceleration for breaking and accelerating. The control then tests the following:

```
(omega_w > omega_e -> (a < 0 & T <= (omega_w - omega_e)/a | a >= 0)) &
(omega_e > omega_w -> (a > 0 & T <= (omega_e - omega_w)/a | a <= 0)) &
```

```
theta_e + omega_e*T > theta_w + omega_w*T + a*T^2/2 &
360 > theta_e + omega_e*T - (theta_w + omega_w*T + a*T^2/2));
```

This control checks 4 main things. The first check `(omega_w > omega_e -> (a < 0 & T <= (omega_w - omega_e)/a | a >= 0))` is to ensure that if Wall•E is moving faster than Eve and is breaking, that he does not break for longer than it the time it takes to get to Eve's velocity. This is so that he doesn't break too far and end up hitting Eve in the other direction. The second is a similar test: `(omega_e > omega_w -> (a > 0 & T <= (omega_e - omega_w)/a | a <= 0))` which tests the same idea: when Wall•E is moving slower than Eve, we want to ensure that he doesn't accelerate too fast so that he then crashes into Eve. The last two checks in the control `(theta_e + omega_e*T > theta_w + omega_w*T + a*T^2/2 &360 > theta_e + omega_e*T - (theta_w + omega_w*T + a*T^2/2))` ensure that given that Wall•E accelerates for time T, he hasn't passed or hit Eve at the end and also that he doesn't fall more than 360° behind Eve (meaning Eve ran into him from behind). If Wall•E accelerates such that he is more than 360° away from Eve, then Wall•E and Eve can be at the same spot, but have different theta values (Wall•E hits Eve, but at by going around the circle one more time than Eve does).

To prove this property, we use the following loop invariant. `0 < theta_e - theta_w & theta_e - theta_w < 360`. This loop invariant helps to ensure that Wall•E and Eve are always within 360° of each other, for two reasons. First, since they're on a circle, Wall•E and Eve could collide if `theta_e == theta_w ± 360°`, or any multiple of 360°. This would mean verifying that `theta_e != theta_w` is useless, since there are many other ways they can collide. Second, if it is ever the case that they are not within 360° of each other, it means that one of them 'lapped' the other, which would mean they collided at some point, which violates our safety condition.

Evaluation the System:

There are two aspects of our system that are useful to analyze: how applicable it is, and how efficient it is. Applicability would measure how close our system matches real world situations. Is it easy to adapt? How well does it model the system? Efficiency would measure how well the system is achieving the goal. Even if it's safe, it could be doing so without accomplishing much. For example, if our goal was to design a system where a moving robot doesn't run into anything, it could just sit still. That would be safe, but not particularly efficient or interesting.

In our case, efficiency is pretty straightforward. Because we verified that Wall•E is neither too close or too far behind Eve, it's guaranteed to be efficient, when efficiency is defined as not being left behind by Eve. We only verified that Wall•E and Eve don't collide and our efficiency condition became an invariant of our system needed to prove safety. A more efficient condition would be to verify that not only do they avoid collisions, but they spiral nicely (meaning Wall•E would always be roughly across the circle from Eve). This actually would not be any harder to verify, since all we'd have to do is change the 0° and 360° bounds in our loop invariant to be more restrictive. For instance, we could prove that instead of `0 < theta_e - theta_w & theta_e - theta_w < 360`, the difference in their thetas is between 170° and 190°, thus showing they are approximately across from each other on the circle.

This of course would mean we would have to change the controller accordingly, but

```
    theta_e + omega_e*T > theta_w + omega_w*T + a*T^2/2 &
360 > theta_e + omega_e*T - (theta_w + omega_w*T + a*T^2/2))
```

is really the same thing as:

```
0 < theta_e + omega_e*T - (theta_w + omega_w*T + a*T^2/2) & theta_e +
        omega_e*T - (theta_w + omega_w*T + a*T^2/2)) < 360
```

So, we could just change the 0° and 360° appropriately and impose more constraints on acceleration and Eve's velocity to prove this new property. So if were to instead use thetas appropriately constrained, and prove that, it would be more efficient.

Another possible issue with our design is that we enforce these two restrictions:

```
(omega_w > omega_e -> (a < 0 & T <= (omega_w - omega_e)/a | a >= 0)) &
(omega_e > omega_w -> (a > 0 & T <= (omega_e - omega_w)/a | a <= 0))
```

These aren't actually required for a controller to be safe, since it might be safe at some point to brake when you start out going faster than Eve, and would end up going slower than Eve. Currently, we restrict our breaking acceleration so that Wall•E never crosses Eve's velocity. We chose to add these because those cases would make it a lot harder to analyze and prove, for instance, when Wall•E is going faster than Eve, he might brake, but in doing so slows down to the point where Eve runs into him.

While our system is fairly efficient, it's not very applicable. Since we didn't constrain the braking or accelerating power, it's not a very realistic model. Infinite braking power would allow Wall•E to approach to just behind Eve and then brake really hard, which would not be safe if Wall•E's brakes weren't strong enough (since a fire extinguisher does have a limited braking capacity). An alternate controller that would constrain the maximum acceleration and the maximum braking to A and B respectively, would better match Wall•E's abilities. We have a draft of a controller that would do this, attached in appendix A, though we weren't able to prove the system. Actually, since all Wall•E has is a fire extinguisher which is either on or off, it might be more realistic to constrain the controller to be able to choose only between accelerate, brake, or coast, as well as have the braking and acceleration be the same value. However, if we consider the three dimensional case, he could tilt the angle and point some of the acceleration in the z-direction, lessening his acceleration around the spiral, so it's not really necessary to only have three values for Wall•E to choose from.

It's also a little unrealistic because when we chose to use polar coordinates, we assumed Wall•E and Eve were centered at the origin. It would not be trivial to adjust our controller to be centered somewhere else, but our proof at least shows that it would be possible. Converting to rectangular coordinates would be possible, but would complicate the differential equations. The proof would then rely on many differential invariants and differential cuts rather than solving the differential equations.
Clearly there are some tradeoffs when designing such a model. Some of the trade-offs were listed above. For example, choosing polar coordinates was a tradeoff between usefulness of the model and ease of proof. We also chose to start with unbounded braking and acceleration because that was easier to prove, though again, it's not as accurate.

Another trade-off was the ease of simulation versus the freedom of the controller. Because we used non-deterministic or star assignment `(a := *)` rather than if-then-else (non-deterministic choice) and deterministic assignment, it became harder to simulate. So if we wanted to simulate our system and look for bugs or possible edge cases, it would be harder to assign a safe acceleration than if we had only a couple conditions to meet and had a deterministic assignment. We chose to do non-deterministic assignment because although it is harder to simulate, it allows more freedom in the controller to choose anything that would be safe, so models a wider range of possibilities than constraining Wall•E to only chose from a limited number of possible accelerations. It also ends up being easier to prove this way since the constraints we then impose on the acceleration are exactly the ones we'd need to use to prove our safety property.

Possible Future Directions

As noted above, there are many possible directions you could take this problem in. We feel there are three major improvements that would be useful and should be feasible. The easier one is to constrain Wall•E acceleration to within a maximum acceleration and maximum braking, as seen in the a proposed controller in appendix A. This would be a more useful and realistic controller, and we have gotten part way through the proof. It seems like it should be possible, but there are many branches and it's hard to tell if one will provide a counterexample.

The other possible improvement is to do what our initial plan was, and model this using a rectangular coordinate system. We moved away from that direction since both the controller and proof get significantly more complicated. You have to use approximations of distance since we cannot use pi, cos, and sin which makes it hard to model circular motion. The differential equations would not be solvable, and consequently the proof would rely heavily upon differential cuts and invariants. We believe that our proof is a precursor to modeling that system, since it may be easier to translate a system that works into rectangular coordinates than it would to start from scratch in that coordinate system.

The third improvement we wanted to make was to control Wall•E's motion in the z-direction as well. As we mentioned, Wall•E is able move the fire extinguisher in the z-direction, thus decreasing his velocity in the circle, while also making him accelerate in the z-direction. This would help increase the applicability of the model as the model would allow for moving in all three directions and show how Wall•E can move in 3-dimensional space.

Appendix A: Proposed bounded controller

This controller we think will model correctly Wall•E keeping up with Eve in polar coordinates such that Wall•E's acceleration is bounded below by -B and above by A (maximum braking and acceleration).

```
\programVariables{
  R A; /* walle's maximum acceleration */
  R B; /* walle's maximum braking */
  R T; /* Time-trigger limit on evolution */
  R t;  /* time */

  R theta_w; /* Position of walle in y direction */
  R z_w; /* Position of walle in z direction */
  R dz_w; /* Unit vector in direction of travel, z direction */
  R r_w; /* Over-approximation on radius of walle */
  R omega_w; /* Angular velocity of walle */
  R a; /* Linear acceleration of walle */

  R theta_e; /* Position of eve in y direction */
  R z_e; /* Position of eve in z direction */
  R dz_e; /* Unit vector in direction of travel, z direction */
  R r_e; /* Over-approximation on radius of eve */
  R omega_e; /* Angular velocity of eve */

  R cute_r; /* Distance between center of walle and center of eve */

  R time_to_brake;
  R dist_to_brake;
  R time_to_acc;
  R dist_to_acc;
}

\problem{
(
omega_w >= 0 & omega_e > 0 &
0 < theta_e - theta_w & theta_e - theta_w < 360 &
T > 0 &
r_w > 0 & r_e > 0 &
dz_w > 0 & dz_e = dz_w & B > 0 & A > 0 &
(omega_w > omega_e -> T <= (omega_w - omega_e)/B) &
(omega_e > omega_w -> T <= (omega_e - omega_w)/A)
)
->
\[(
   a := *;
   time_to_brake := ((omega_w + a*T) - omega_e)/B;
   dist_to_brake := ((omega_w + a*T)^2 - omega_e^2)/(2*B);
   time_to_acc := (omega_e - (omega_w + a*T))/A;
   dist_to_acc := (omega_e^2 - (omega_w + a*T)^2)/(2*A);
   ?( a >= -B & a <= A &
     0 <
```

```
      theta_e + omega_e*T + omega_e*Max(time_to_brake, time_to_acc) -
      (theta_w + omega_w*T + a*T^2/2 + Max(dist_to_brake, dist_to_acc))
      &
      theta_e + omega_e*T + omega_e*Max(time_to_acc, time_to_brake) -
      (theta_w + omega_w*T + a*T^2/2 + Max(dist_to_brake, dist_to_acc))
      < 360);
    t := 0;
    {
      theta_w' = omega_w, z_w' = dz_w, omega_w' = a,
      theta_e' = omega_e, z_e' = dz_e,
      t' = 1, t <= T, omega_w >= 0
     }@invariant(t >= 0)
    )*@invariant(
                omega_w >= 0 &
                0 < theta_e - theta_w &
                theta_e - theta_w < 360 & B > 0 & A > 0 &
                (omega_w > omega_e -> T <= (omega_w - omega_e)/B) &
                (omega_e > omega_w -> T <= (omega_e - omega_w)/A)
                )
\](theta_e != theta_w)
```