15-424/15-624/15-824 Recitation 5
Debugging models, Understanding Failed Proofs, Differential Cuts
15-424/15-624/15-824 Foundations of Cyber-Physical Systems

1 Debugging Models and Proofs

Constructing formal models of complex dynamical systems is difficult, and finding formal proofs about these models is even more difficult. You have learned from your experience programming that it is somewhat rare to write down a program from start to finish without introducing any bugs. Therefore, it should be unsurprising to you that models and proofs will often contain bugs. Identifying the source of the problem – a bad model, a good model with a buggy controller, or a good model and a good controller with a bad proof – is a crucial skill in large CPS proof engineering projects.

We will practice debugging models and proofs using a very simple model of a tank with both a fill rate and a drain rate. Imagine a big tank (really big – no upper bound on capacity!) with some water flowing in at a constant rate and some water flowing out at a rate that you can control. Here's a first attempt at the model:

$$
\begin{array}{l}
level > 0 \land \epsilon > 0 \rightarrow \\
\big[\{ \\
\quad drainRate := *; \\
\quad ?(fillRate - drainRate)\epsilon > 0; \\
\quad t := 0; \\
\quad \{level' = fillRate - drainRate \\
\quad\ \ \&\epsilon > 0 \land level > 0\ \} \\
\}^*\big] level > 0
\end{array}
$$

By now, you should be able to come up with the canonical proof:

```
implyR(1) &
loop({`level>0 & ε>0`}, 1) <(
  QE,
  QE,
  composeb(1) & randomb(1) & allR(1)   & /* randomb handles [:=*] */
  composeb(1) & testb(1)   & implyR(1) &
  composeb(1) & assignb(1) &
  diffSolve(1) & QE
)
```

Proof found, no debugging necessary! But wait – let's replace that `diffSolve` with a `diffWeaken` instead. The resulting (provable!) subgoal looks fishy:

$$\epsilon > 0 \land level > 0 \vdash \epsilon > 0 \land level > 0$$

Having $level \geq 0$ as a domain constraint makes a lot of sense – tanks can't have negative water in them – but our evolution domain constraint was actually just a tad too strong.

After changing the evolution domain constraint to $\epsilon > 0 \land level \geq 0$ the same tactic still proves the property. But after closer inspection of the model itself we notice that the controller is very conservative – in fact, we can *never* actually drain water from the tank! Notice that we choose $drainRate$ such that $(fillRate - drainRate)\epsilon > 0$. But ϵ is always positive, so that means that we never choose a $drainRate$ greater than the $fillRate$. Indeed, our drain is rather useless in this case and the safety property is trivially true. We can see this concretely by pausing the proof right before the final `QE` and (propositionally) cutting in $drainRate < fillRate$ and hiding the original proof goal:

```
implyR(1) &
loop({`level>0 & ε>0`}, 1) <(
  QE,
  QE,
  partial(
    composeb(1) & randomb(1) & allR(1)   & /* randomb handles [:=*] */
    composeb(1) & testb(1)   & implyR(1) &
    composeb(1) & assignb(1) &
    diffSolve(1) & cut({`drainRate < fillRate`}) <(
      QE,
      partial(implyR(1) & hideR(2)) /*QE proves drain<fill*/
    )
  )
)
```

Let's improve the model by trying a slightly less conservative choice of *drainRate*. We still make sure that there is some water left in the tank at ϵ time, but we also take into account the current level of water in the tank:

$$
\begin{array}{l}
level > 0 \land \epsilon > 0 \rightarrow \\
[\{ \\
\quad drainRate := *; \\
\quad ?level + (fillRate - drainRate)\epsilon > 0; \\
\quad t := 0; \\
\quad \{level' = fillRate - drainRate \\
\quad\ \&\epsilon > 0 \land level \geq 0 \ \} \\
\}^*]level > 0
\end{array}
$$

Our original tactic does *not* prove this model is correct, so let's modify our original tactic to pause right after `diffSolve` so we can find a counter-example:

```
/* Reduce water tank to arithmetic */
implyR(1) &
loop({`level>0 & ε>0`}, 1) <(
  QE,
  QE,
  partial(
    composeb(1) & randomb(1) & allR(1)   & /* randomb handles [:=*] */
    composeb(1) & testb(1)   & implyR(1) &
    composeb(1) & assignb(1) &
    diffSolve(1)
  )
)
```

Running "find Counter-example" on the resulting arithmetic results in the following counter-example:

$$
\begin{aligned}
drainRate &= 1 \\
fillRate &= 7/8 \\
\epsilon &= 7/8 \\
level_0 &= 1/8 \\
t_{-0} &= 0 \\
level &= 0 \\
t_{-} &= 1
\end{aligned}
$$

Notice two fishy things:

- $t\_ = 1$ but $\epsilon = 7/8$
- t isn't mentioned anywhere in the arithmetic – only $t\_$. This is subtle! $t\_$ is a totally different variable from t; Variable names that end with underscores are almost always *built-in*. In this case, $t\_$ is a built-in variable that represents the true time that's passed in the ODE, while t is our explicit clock variable.

Let's modify the model to use a less confusing variable for the clock (e.g., Z). Also, based upon the amount of time the system ran, it's not obvious that we've forgotten to give physical dynamics to our clock and also forgotten to make sure that ϵ is an upper-bound on the amount of time the system runs. Both of these problems are easily fixed:

$$
\begin{array}{l}
level > 0 \land \epsilon > 0 \rightarrow \\
\big[\{ \\
\quad drainRate := *; \\
\quad ?level + (fillRate - drainRate)\epsilon > 0; \\
\quad Z := 0; \\
\quad \{level' = fillRate - drainRate, Z' = 1 \\
\quad\ \&0 < \epsilon \land Z \leq \epsilon \land level \geq 0 \ \} \\
\}^*\big] level > 0
\end{array}
$$

The original tactic works on this model! But is the model sensible? To see that it isn't, run the "reduce to arithmetic" partial tactic from above on this model and then cut in $drainRate < 0$. Since we *subtract* the drain rate in the dynamics, a negative $drainRate$ actually means the tank is *filling* with water from underneath! This should never be the case. Therefore, we should probably at least force $fillRate \geq 0$ and maybe also test that $drainRate \geq 0$. If we do so, then $fillRate \geq 0$ will have to be added to the loop invariant.

We could do a lot to make this model more realistic. In this model we're assuming that we can directly measure the $fillRate$ and directly control the $drainRate$. In reality, we would probably only have control over the size of the opening for the drain. And we would probably have to compute the fill rate based upon the size of a pipe and other variables. We'll leave building a more realistic water tank to future recitations, though.

Bugs in Proofs KeYmaera X is designed so that it is hard or impossible to write down an incorrect proof. However, not even KeYmaera X can prevent us from making mistakes in a proof. For example, suppose we want to prove:

$$x > 0 \rightarrow [x' = 1]x > 0$$

The proof of this property is pretty simple; e.g. `implyR(1) & diffSolve(1) & QE` will do just fine. However, if we weaken the differential equation instead of solving the differential equation then we have to prove:

$$x > 0$$

which is actually quite hard to prove. Differentiating between proofs that fail because the model does not meet its specification and proofs that fail because there is a flaw in the proof is an important skill that we will continue to develop for more complicated models in future recitations. And you're sure to get some experience debugging proofs in your labs!

Conclusions

- Models may satisfy safety properties even when the model is a bad model of the system.
- A failed proof doesn't necessarily mean that KeYmaera X is broken! Sometimes, the proof is structurally correct but the arithmetic doesn't work out at the end because the model *really is* wrong.
- Similarly, counter-examples in arithmetic don't necessarily mean that the model is wrong. It could just be that the particular proof technique you've chosen doesn't work for the model at hand.

2 Differential Cuts and Differential Invariants

We are starting to encounter systems with non-linear dynamics, and going forward we will have to prove properties about systems with progressively more complicated physical dynamics. For instance, in labs 3 and 4 you will verify models of systems with circular dynamics. Although these dynamics are solvable, their solutions are not expressible in the decidable theory of real arithmetic implemented by Keymaera X. Also, in general, most ODEs are not solvable.

When a system's physical dynamics are not possible to reason about using a solution, we have to use invariants to establish properties about the system's behavior. Invariants and cuts sounds scary, and rightfully so – their proof theory is very deep and subtle. But as it turns out, you are far more familiar with these concepts than you thought! As a warm-up, consider a typical linear system augmented with an explicit time variable function:

$x = 0 \land v = 0 \land a = 1 \rightarrow \{x' = v, v' = a, t' = 1\}$

We can establish something like the result of `diffSolve` using differential cuts:

```
implyR(1)                                  &
diffInvariant({`t >= 0`}, 1)               &
diffInvariant({`a=1`}, 1)                  &
diffInvariant({`v=a*t`}, 1)                &
diffInvariant({`x=(a*t^2)/2`}, 1)          &
diffWeaken(1)                              &
QE
```

Observe that you have actually been thinking about invariants ever since you learned about ODEs – solving a linear ODE is essentially just establishing a sequence of differential invariants about a system with an explicit time variable.

It is not always possible to establish a desired invariant in one step. For example, trying to establish $x = \ldots$ as an invariant before first establishing $v = at$ as in invariant eventually requires proving the validity of

$$v = \frac{(4at - at^2)}{4}$$

which is obviously not valid[1].

Of course, invariants are only interesting when they are necessary – namely, in situations where an ODE cannot be solved or its solution is not expressible in KeYmaera X. For example, recall the final problem on the Midterm:

$$x \geq 2 \land y \geq 22 \rightarrow [x' = 4x^2, y' = x + y^4](y \geq 22).$$

If we try `diffInd` directly then we have to establish that

$$[y' := xy^4][x' := 4x^2])(y \geq 22)$$

By `derive` and `Dassign` it would suffice to show

$$x + y^4 \geq 0$$

which is rightfully hard to prove in KeYmaera X only because we have no constraints on the value of x. But *we* know that $x \geq 0$ must be true throughout any flow of this differential equation. Establishing this first adds $x \geq 0$ to our domain constraint, which we then get to assume when establishing other invariants of the system (in this case, the post condition):

```
implyR(1)                          &
diffInvariant({`x >= 0`},1)        &
autoDiffInd(1)
```

[1] You can see how this subgoal arises by using `diffCut` followed by `diffInd`