Research Plan

What would make a unsat core tool usable?

1. For a lot of specifications, find conflicts
2. Maybe pointing out when specifications are unnecessairly complex...
3.

Create benchmarks for LTL and MLTL
LTL I will use SPOT.
MLTL I will use WEST?

The goal is to end up with a robust set of unsat formulas.
by "Robust", I mean over 200 formulas for one "Example", similar to eurail, but open source.

I think I want about 3 unsat examples for the datasets, based on common formulas (from Rozier's work and the past presentation I made, in addition to any open-source ltl requirement databases I can find (Schuppan has an elevator example)).

Use that database to create/test my tool?

Use that database to test understandability of my visuals.

Use that database to exemplify the importance of the unsat cores (remember the reasons that Chris said for planning).

Current Work

Mini-Sat Solver

I'm building a mini-sat solver to be included in the final unsat core tool. The purpose of building my own if for 2 reasons. 1: I would like to know how sat solvers work so I can use them in the future. 2: Having an internal sat solver is better for "download and run" usability because it packages everything up in one place. (may wrap a sat solver api in the future, but this is what I'm doing right now.)

0.0.1 Features

- Accept a CNF formula input
- Parse input into an internal structure
- Run basic DPLL style sat solver
- display whether the formula is SAT or UNSAT
- if SAT: show a model
- if UNSAT: Show unsat core

0.0.2 Pseudocode, Logic, Proof of Correctness

Mini-Sat solver will

1. parse into clauses
2. identify the variables
3. generate **every** possible truth assignment (yeah, I know...)
4. Check if there exists one assignment that makes every clause True
5. if yes → Sat assignments. If no → "unsat"
    - the unsat core will be in a different algorithm

Algorithm 1 Mini-Sat Solver

Input:CNF & NNF Algorithm with less than 3 variables and only !,&,|
Output: Satisfiable Assignments or an Unsat Core

Algorithm 2 Parse Formula

Input:CNF & NNF Algorithm with less than 3 variables and only !,&,| ▷ ex. "(A | B) & (!A | C)"
Output: Nested list of all variables in each clause
clauses = array of clauses split on & ▷ ex. [["A | B"], ["!A | C"]]
variables = array of variables split on | ▷ ex.[["A","B"],["!A","C"]]
return variables

Algorithm 3 Get Variables

Input: Nested List of Variables ▷ ex.[["A","B"],["!A","C"]]
Output: List of unique variables ▷ ex.["A","B","C"]
unique_variables = []
while original list is not empty **do**
    temp_variable = curent variable in list
    if temp_variable starts with "!" **then**
        remove "!"
    end if
    if Variable not in unique_variables **then**
        Add Variable to unique_variables
    end if
end while
return unique_variables

design decisions

- React Framework, common and usable across all platforms. Other Devs can modify it without too much additional learning
- Input type, same as java and javascript because developers will be used to similar style of logical equivalance
  - —— or
  - && and
  - ! not
  - == equal
  - != not equal
- input type will be expanded to accept single words (python style) and single characters '&' instead of '&&' so people can write their formulas however they want
- I will not allow implies and iff, but I will provide equivalances in the "how to" menu for people to convert with.
- I am rejecting more than 3 variables and any formula that is not in CNF

Done

- React App created
- React App takes input and outputs the same input :)
- Backend for mini-sat solver is created (no logic, but it's running :))
- Connection from the back end to the front end when submitting a formula

0.0.3 Working on

- Basic Sat Solving Logic

Introduction

It is well known that early system design is a critical moment in the development process. Part of that early-stage system design is requirements elicitation and debugging. During this time, engineers and designers don't have a full system model yet, making requirements debugging difficult, and model checking impossible. It is widly recognized that bugs not found in early stages of development will exponentally increase cost of repairing as the project progresses. So it's critical to write, check, and debug your requirements as early as possible.

Industries like aerospace, trains, robotics, other safety-critical industries systems need to respond over time. Example: If this button is pressed, the train

door should close within 5 seconds" (use a better safety critical example, possibly one of roziers Air traffic control requirements), which is exactly what Linear Temporal Logic is built for.

[add something to introduce LTLf, MTL, and MLTL]

Current requirements validation:

- Model checking (if the system model satisfies the requirements), requires a model (what the system does) and the requirements(what it should do).
- Vacuity Checking, so that your requirement isn't trivially satisfied. (If A happens, then B should happen, and A never happens) ¡- satisfied, and maeningless.
- Coverage: Measures how much of the model is actually tested by the requirements

What do we do when we don't have a model?

Satisfiability & Realizability **satisfiability:** *Are these requirements even logically* ***possible*** *to satisfy?*

Realizability *Is there a way to build a system thay will always follow the rules?* Both of these checks work with requirements and without a system model.

The Problem

Requirements can be unsatisfiable Checking requirements of real systems by hand is impossible because of their size (Find some examples of systems by looking up why requirements checkers were made)

Satisfiability (SAT)

SAT: *Assigning True or False to variables so that a whole logical formula becomes True.*

unsatisfiability: *All possible combinations of True and False for all variables result in the formula being False.*

Relevance to My Paper Sat and Unsat ask if there is a contradiction. Unsat for LTL asks the same question but is more complex because it is over time. Unsat for LTLf is more complex because it reasons over finite LTL. Unsat for MLTL asks the same question but is more complex because it reasons over specific time steps.

SAT Solving

Methods involve **Clever search and pruning**

Unsat Cores

Unsatisfiable Core: *A part of a formula that makes satisfiability impossible.*

- Useful for:
  - **Debugging**
  - **Faster checking** in model checkers
  - **Certifying** that something is truly unsatisfiable

Linear Temporal Logic

My Ideas

The enitre project is about usability
Internal SAT Solver - So people don't have to download their own sat solver. (Part of "download and run" capabilities)

I'm going to publish guidelines for making your formal methods tools usable. If your tools meets these standards, you get to have a mark of approval :)

1. MLTL Syntax trees SYNTAX TREES DO NOT CAPTURE TIME IN ANY REASONABLE WAY, SCHUPPAN IS AN IDIOT, JUST TRY DOING IT WITH UNTIL A possible research insight would be why U and R break syntax trees.... If I want Schuppan to hate me
   (a) That being said, Could still probably do a syntax tree definition form MLTL while acknowledging the lack of ability to use it for anything more complex than F and G
   (b) Unless, you can express until and release in terms of F and G
2. Sat Solvers can already extract a core subset of the clauses (unsat cores)

For usability, we need to determine if a (set of) requirements is satisfiable to begin with. This may already be possible with model checkers (for MLTL?)

Representation of Unsat Cores

Given the formula

$$\phi = (G(p \wedge q)) \wedge (F(\neg p \wedge r))$$

The main components that make the core unsatisfiable are

$$[p, \wedge, \neg p, G, F]$$

These components by themselves are not indicitave of unsatisfiability, because we could have

$$G(F(p) \wedge F(\neg p))$$

Which is satisfiable. The important part is how the components are ordered.

UC as sets of sub formulas

Would be presented as the sub formula

$$SF(phi) = G(p \wedge q) \wedge F(\neg p \wedge r), G(p\ wedgeq), F(\neg p \wedge r), p, \neg p, q, r$$

and the sub formulas that make the unsat core possible

$$UC(phi) = G(p \wedge q) \wedge F(\neg p \wedge r), G(p\ wedgeq), F(\neg p \wedge r), p, \neg p$$

The problem with this method, is that it is not immediately understandable and by examination of this simple unsat problem, would not scale very well.

UC as reduced versions of the original formula

This option would present the unsat core as a version of the formula that is not logically equivalent to the original formula, but more immediately points out the cause of the unsatisfiability.

$$UC(\phi) = G(p) \wedge F(\neg p)$$

The problem with this is that it does not give context of where the unsatisfiable parts are located in a formula. This may not be scalable, but that might not be a problem, depending on how formulas are written in practice.

UC as color coded unsatisfiability

$$UC(\phi) = (G(p \wedge q)) \wedge (F(\neg p \wedge r))$$

Which literally highlights the parts of the formula that are causing unsatisfiability, while giving context for the location of unsatisfiability. I like this one because it keeps equivalance, and highlights problems.

The problem is that if there are multiple variables or conjuntions that cause unsatisfiability, how do you enumerate that?

A possible solution could be to create a UI where you can "show all" problems, or toggle "show one problem at a time" with buttons to switch between problems.