Specification and Analysis of Requirements (SpeAR)

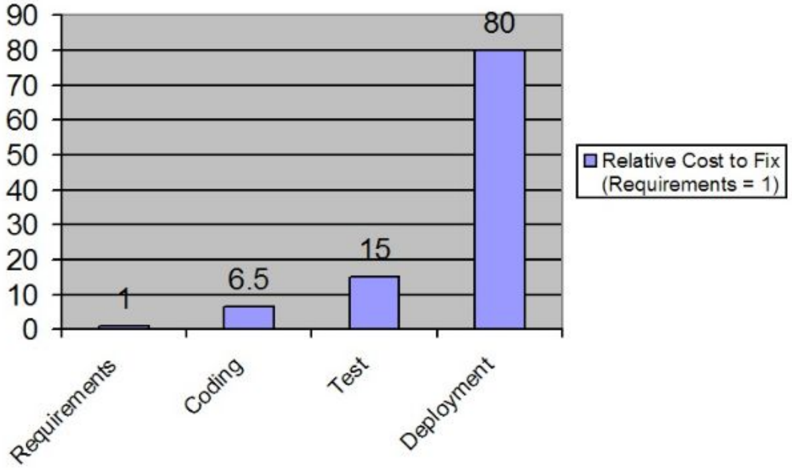
Building trust every day

Requirements: They're hard

Developing correct requirements is difficult to do in natural language: Some ways to evaluate NL requirements:

- peer review (not rigorous)
- prototyping (could be expensive)
- natural language processing (good at finding bad "smells", but not deep behavioral issues, at least yet)

Requirements mistakes in later development is costly; especially in commercial avionics systems.

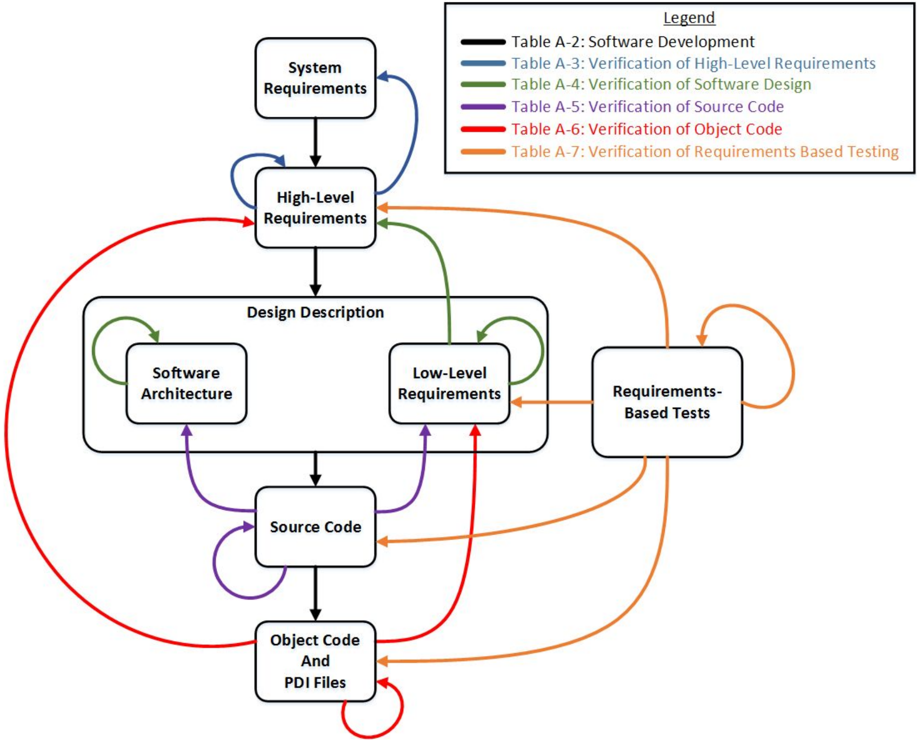


Rockwell Collins

Example: Avionics

Requirements mistakes touch every downstream artifact.

- Artifacts must be updated.
  - HLR
  - Design
  - Source
  - Object code
  - Tests
- Verification results must be re-established.
  - Peer reviews
  - Testing
  - Analyses (Coverage)



Idea!

Why not model functional requirements mathematically? To start, consider:

- Critical (safety or security) functionality
- Complex logical behaviors

Avoid:

- Nonfunctional requirements (for example, the “ilities”)

Formally modeling requirements provides benefits.

- Requires greater care and thought
- Assigning semantics improves comprehension across teams

Use formal methods to establish correctness of requirements.

- Exhaustive analyses find errors.

Formal Methods: What is it?

From: en.wikipedia.org/wiki/Formal_methods

> "Formal methods are a particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems."

Typically, formal methods refers to the following three techniques:

- Abstract Interpretation
- Model Checking and Automated Theorem Proving
- Deductive Theorem Proving

Model Checking

From: en.wikipedia.org/wiki/Model_checking

> "Given a model of a system, exhaustively and automatically check whether this model meets a given specification."

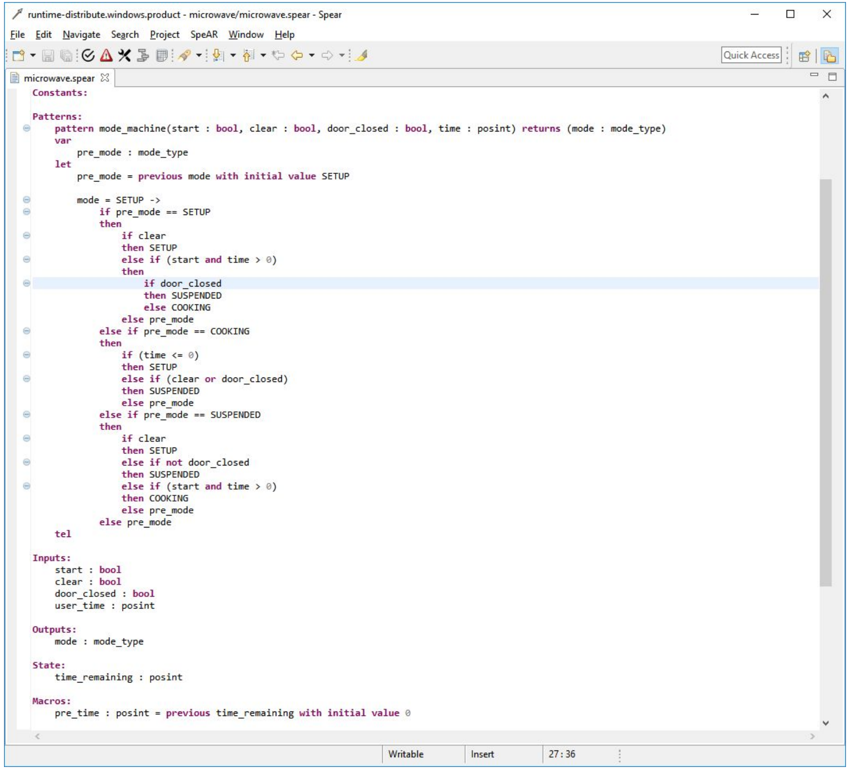
Could we use model checking to rigorously analyze a set of requirements to check for:

- Correctness with respect to a set of properties
- Consistency (Is there a satisfying trace?)
- Realizability (Are there any conflicting traces?)

SpeAR: What is it?

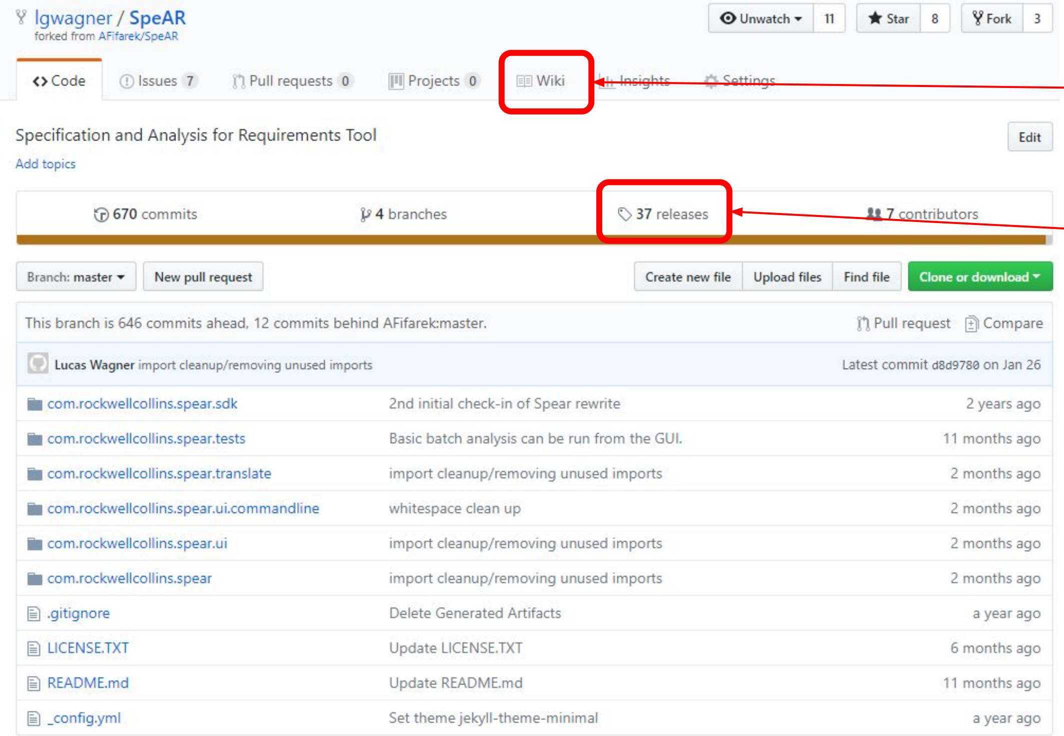
SpeAR is an environment for formally specifying and analyzing requirements. What does that mean?

- Formal: the requirements are backed by formal semantics. i.e. they are precise.
- Analysis: we want to use these formal semantics to enable tools to do deep analysis of the requirements.



SpeAR Information

Repository: github.com/lgwagner/SpeAR



Wiki here!

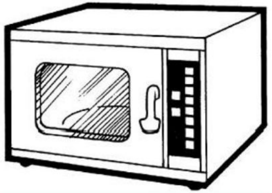
Tool installation files here!

SpeAR Example: A simple microwave oven

Let's design a SpeAR specification for a (very) simple microwave oven.

Our (simple) interface

- Inputs
  - start
  - clear
  - door_closed
  - user_time
- Outputs
  - mode
  - cook_time



What are some desired behaviors of this microwave?

- Cooks when the user presses start and the door is closed.
- Stops cooking when the door opens, the cook time expires, or the user presses clear.
- Remembers cooking time remaining when stopped by the user.
- Anything else?

Microwave Interface

```
Types:
    mode_type : enum { SETUP, COOKING, SUSPENDED }
    nonneg : { t : int | t >= 0 }

Inputs:
    start : bool
    clear : bool
    door_closed : bool
    user_time : nonneg

Outputs:
    mode : mode_type

State:
    cook_time : nonneg
```

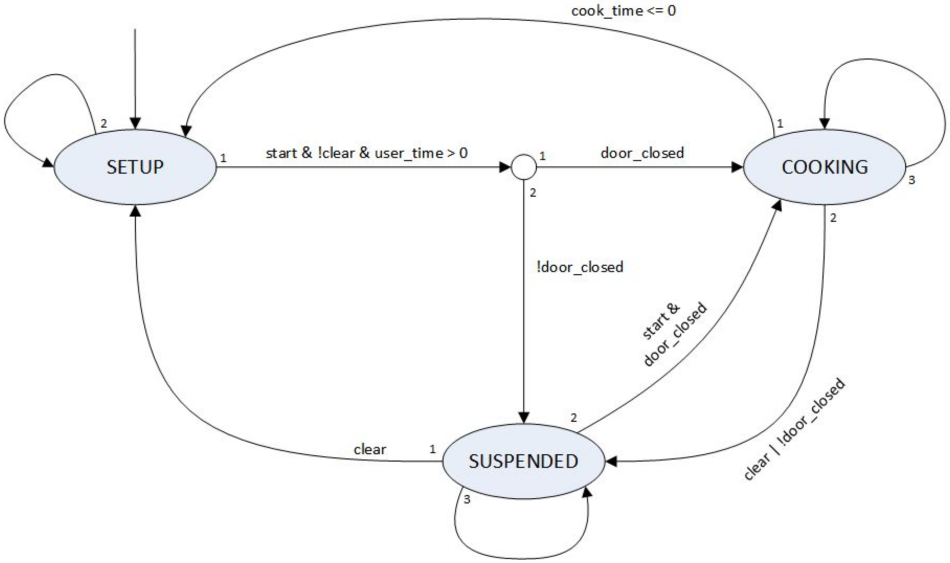
- Types
  - enumeration for microwave mode
  - predicate subtype for time called nonneg or “non negative”.
- Variables
  - Monitored (Inputs)
  - Controlled (Outputs, State)

Microwave: Macros, Assumptions, and Requirements

```
Macros:
    pre_cook_time : posint = previous cook_time with initial value 0
    pre_mode : mode_type = previous mode with initial value SETUP

Assumptions:
    //start and clear will not be pressed simultaneously
    a0: not (start and clear)

Requirements:
    //mode is defined by the state machine implemented in pattern "mode_machine"
    r0: mode == mode_machine(start,clear,door_closed,user_time,cook_time)

    r1: if mode == SETUP
        then cook_time == 0
        else if (pre_mode == SETUP) and (mode <> SETUP)
        then cook_time == user_time
        else if (pre_mode == COOKING) and (mode == COOKING)
        then cook_time == pre_cook_time - 1
        else cook_time == pre_cook_time
```


Rockwell Collins

Microwave Patterns




```
pattern mode_machine(start : bool, clear : bool,
                     door_closed : bool,
                     user_time : posint,
                     cook_time : posint)
    returns (mode : mode_type)
var
    pre_mode : mode_type
let
    pre_mode = previous mode with initial value SETUP

    mode = SETUP ->
        if pre_mode == SETUP
        then
            if (start and not clear and user_time > 0)
            then
                if door_closed
                then COOKING
                else SUSPENDED
            else pre_mode
        else if pre_mode == COOKING
        then
            if (cook_time <= 0)
            then SETUP
            else if (clear or not door_closed)
            then SUSPENDED
            else pre_mode
        else if pre_mode == SUSPENDED
        then
            if clear
            then SETUP
            else if (start and door_closed)
            then COOKING
            else pre_mode
        else pre_mode
tel
```


Microwave Properties and Observers

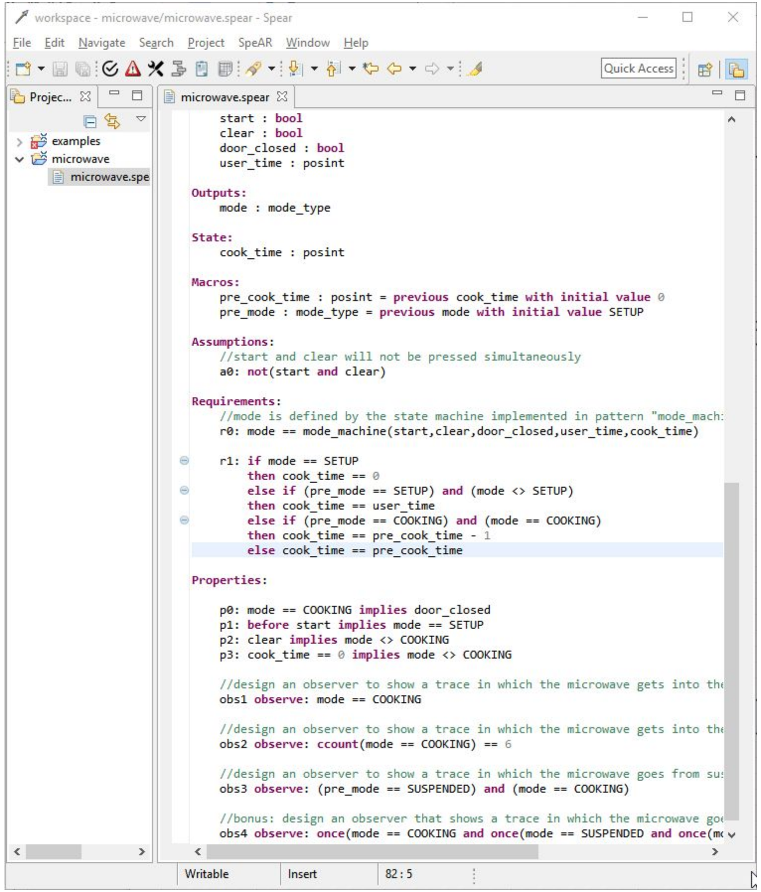
```
Properties:
  p0: mode == COOKING implies door_closed
  p1: before start implies mode == SETUP
  p2: clear implies mode <> COOKING
  p3: cook_time == 0 implies mode <> COOKING

  //design an observer to show a trace in which the microwave gets into the cooking mode
  obs1 observe: mode == COOKING

  //design an observer to show a trace in which the microwave gets into the cooking mode for 6 consecutive steps
  obs2 observe: ccount(mode == COOKING) == 6

  //design an observer to show a trace in which the microwave goes from suspended to cooking
  obs3 observe: (pre_mode == SUSPENDED) and (mode == COOKING)

  //bonus: design an observer that shows a trace in which the microwave goes from cooking to suspended to cooking
  obs4 observe: once(mode == COOKING and once(mode == SUSPENDED and once(mode == COOKING)))
```


Microwave: Entailment Analysis

Entailment proves that the set of requirements satisfy all of our properties, exhaustively.

The analysis also identifies any satisfying traces for observer properties.

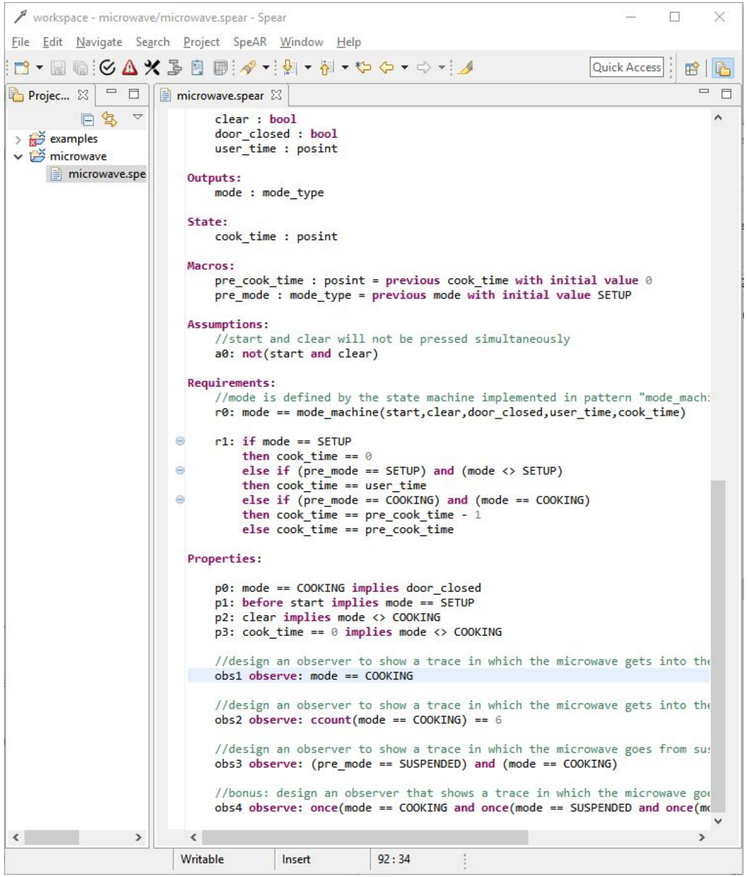
obs2 observe: ccount(mode==COOKING) == 6



Microwave Observers

The purpose of observers is to identify a trace that satisfies the model and the property of interest.

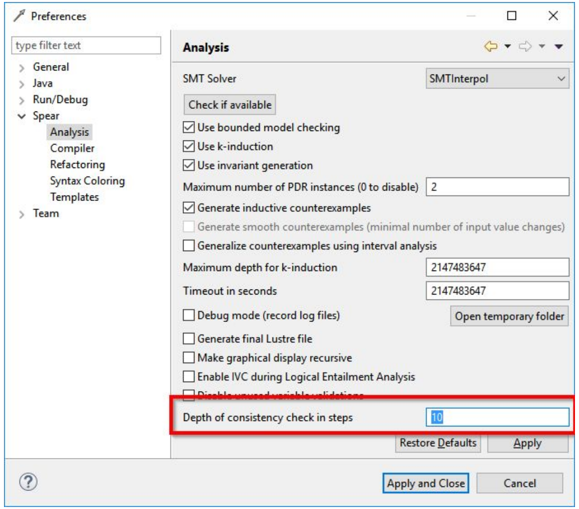
- These traces can help validate that our requirements are correct.
- Can use them as test cases to verify an implementation.

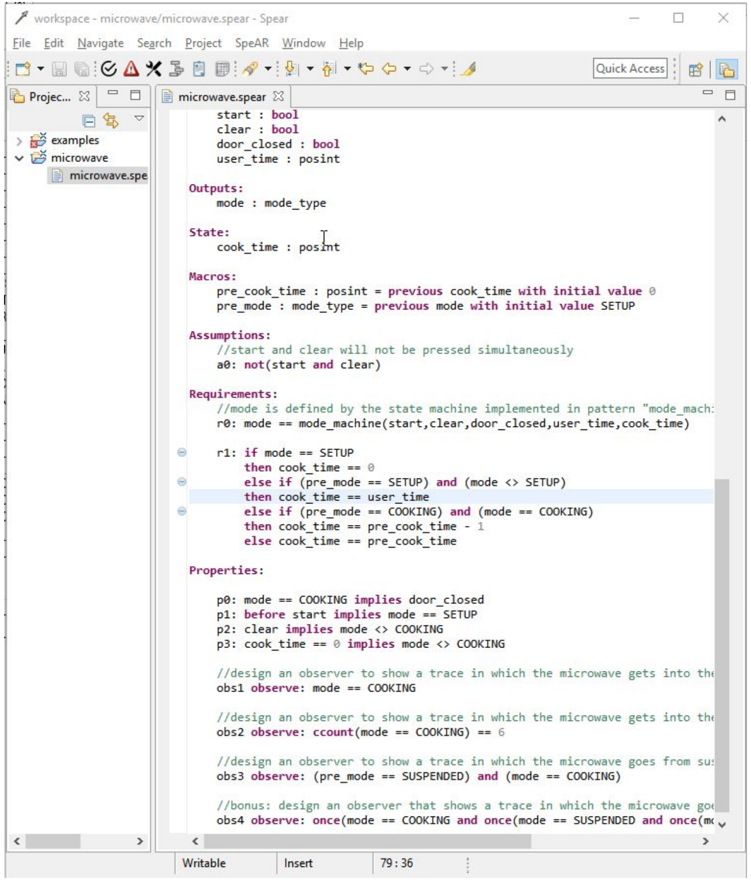


Rockwell Collins

Microwave Consistency

The consistency check tries to identify at least *one* trace that satisfies the model to a depth of N (user configurable).

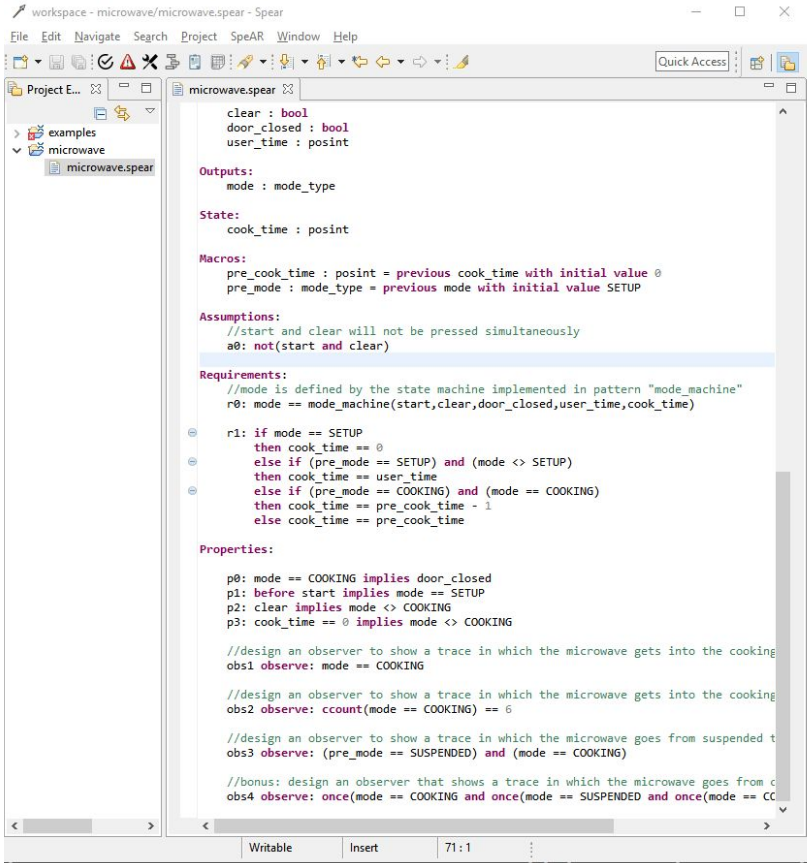




Microwave Realizability

A more powerful (but less scalable) analysis checks if *all* traces are free from conflict.

- If it is able to prove the requirements are conflict free, use it.
- If not, you can rely on the less powerful, but still useful consistency check.



Examples galore...

The Wiki walks through, in depth, two examples.

- Thermostat: github.com/lgwagner/SpeAR/wiki/Overview-of-SpeAR
- A more complex, multi-file microwave: github.com/lgwagner/SpeAR/wiki/Complex-Example

Other examples:

- Turboencabulator software from S5 presentation:
  - Presentation at www.mys5.org/Proceedings/2017/Day_1/2017-S5-Day1_1335_Wagner.pdf
  - Files at github.com/lgwagner/SpeAR/tree/development/com.rockwellcollins.spear.tests/tests/s5

