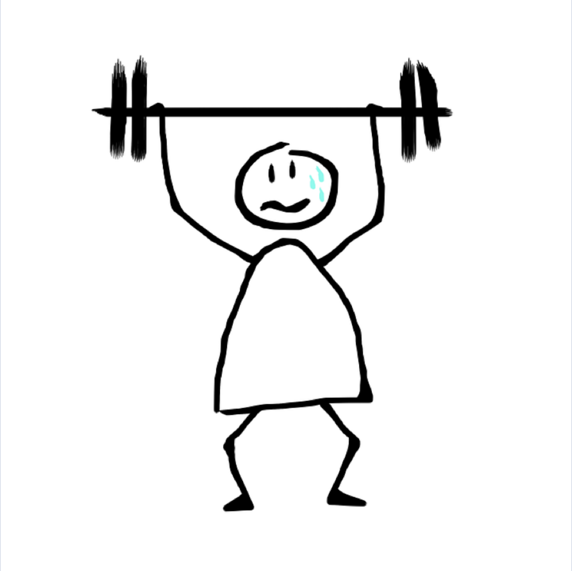
JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

Semi-automated Test Case Generation for ACAS X Implementation Validation

Daniel Genin, Mark Thober, M. Scott Doerrie
Johns Hopkins University Applied Physics Laboratory

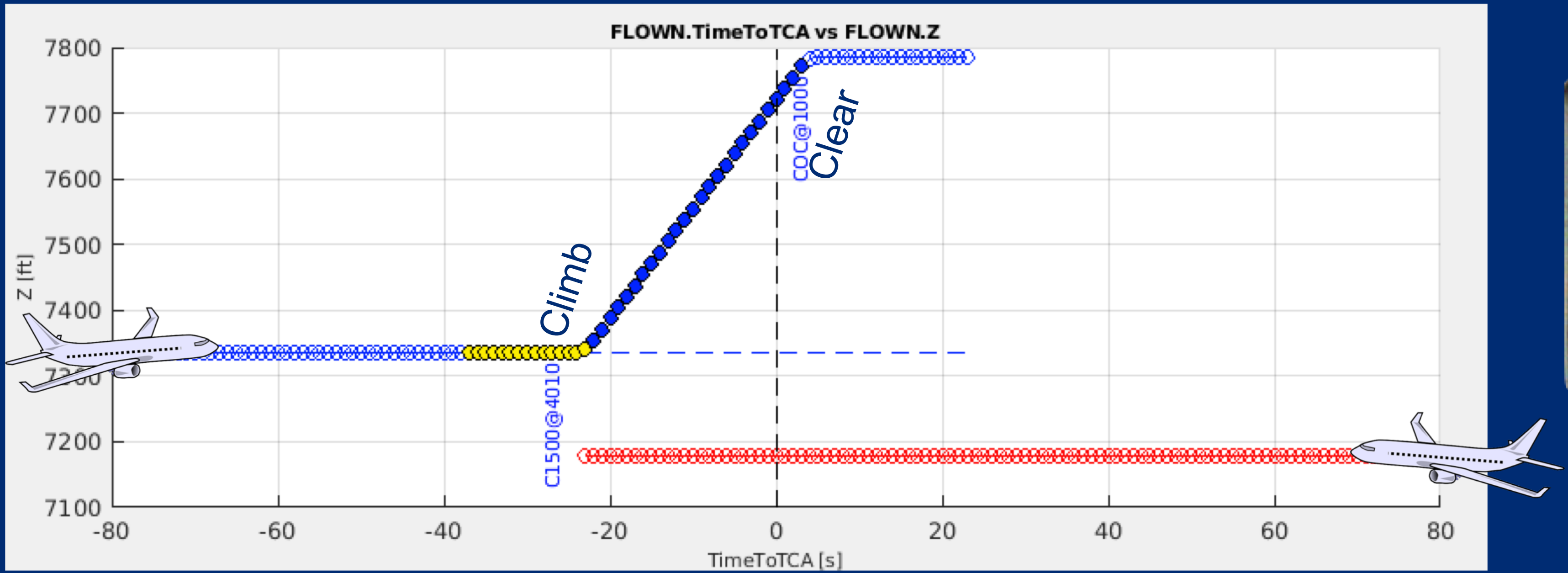
Test case generation

- Test suites are important for development and sometimes mandated
- Test case generation is time consuming and hard
- Software tools for test case generation can dramatically reduce the effort



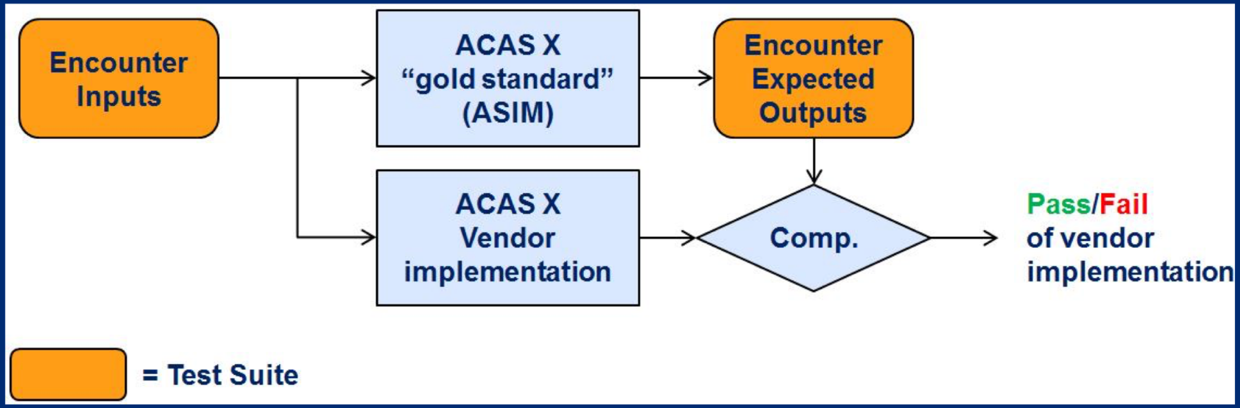
ACAS X

- Airborne Collision Avoidance System X (ACAS X)
  - Next generation replacement for the existing Traffic-Alert and Collision Avoidance System (TCAS)
  - Issues audible and visual collision resolution advisories to pilots, e.g., ”climb now!”



ACAS X Design Process

- Developed by MIT LL and JHU APL under auspices of FAA and in collaboration with representatives of manufacturers and other stakeholders
- RTCA published Minimum Operational Performance Standards (MOPS) as DO-385
  - ACAS X Algorithm Design Document (ACAS X ADD)
  - Implementation requirements
- Manufacturers implement ACAS X ADD and are responsible for certification
  - proprietary implementations
- One of the tools for verifying implementation compliance is the ACAS X Test Suite (included in DO-385)



Encounter Inputs

ACAS X "gold standard" (ASIM)

Encounter Expected Outputs

ACAS X Vendor implementation

Comp.

Pass/Fail of vendor implementation

🟧 = Test Suite

Test Suite

- Test Suite is a collection of end-to-end software tests
  - JSON sensor input sequences (aka encounters) and expected output sequences
- Test Suite provides coverage along several dimensions
  - Output space coverage
  - Functional coverage
  - Branch coverage, i.e. every branch of every conditional statement
    - DO 178C requirement
    - 2128 total branches

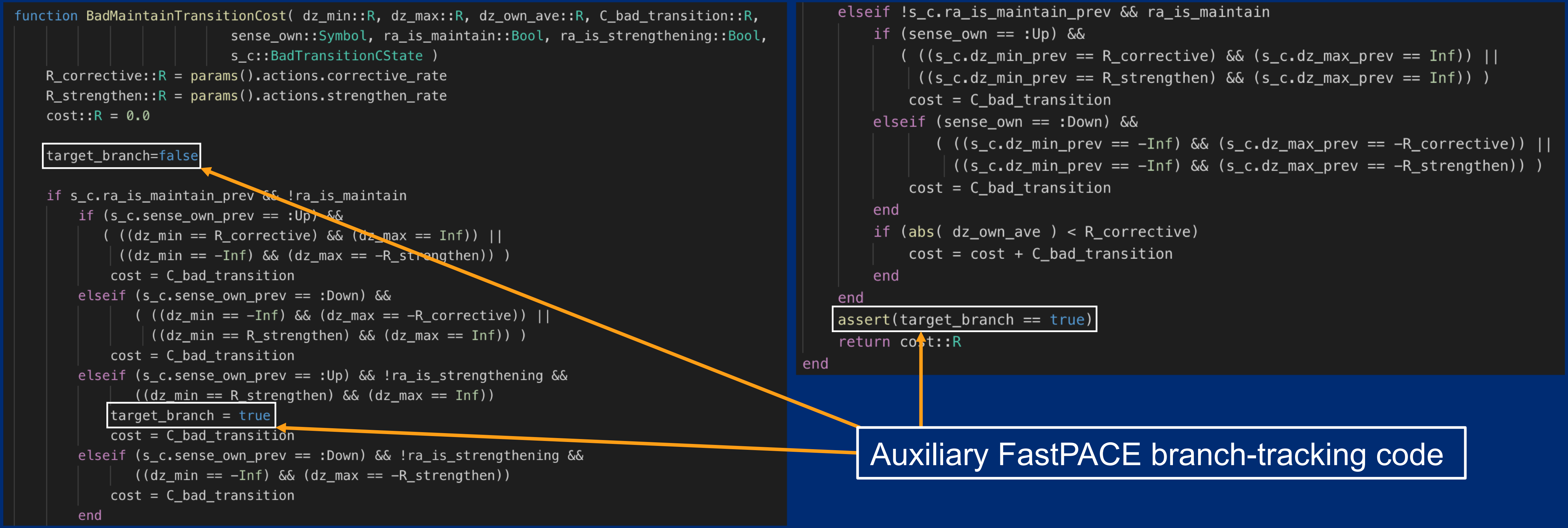
```
{"report_time":0.96,"report_type":"ACAS_XaXo_V15R3","acas_xaxo_v15r3":{"data_type":"OWNSHIP_DISCRETES","ownship_discretes":{"toa":0.96,"address":1,"mode_a":1200,"opflg":true,"manual_SL":0,"own_ground_display_mode_on":true,"on_surface":false,"aoto_on":true,"is_coarsely_quant":false}}},

{"report_time":0.961,"report_type":"ACAS_XaXo_V15R3","acas_xaxo_v15r3":{"data_type":"HEADING_OBS","heading_obs":{"toa":0.961,"heading_true_rad":0,"heading_degraded":false}}},

{"report_time":0.962,"report_type":"ACAS_XaXo_V15R3","acas_xaxo_v15r3":{"data_type":"BARO_ALT_OBS","baro_alt_obs":{"toa":0.962,"baro_alt_ft":5000}}},
```

Branch coverage

- Executable specification allows mechanization of branch coverage at the ADD level
- Julia's powerful Lisp-like macro system allows collection of detailed branch coverage statistics with virtually no changes to ADD code
  - ptf macro
- Significant branch coverage obtained by taking encounters used for simulation tests
  - 500K encounters
  - Greedy branch coverage optimization
  - Still left several hundred uncovered branches

Computer assisted test case development

- Manual test case generation for branches deep in convoluted code is very labor intensive
- FastPACE (Provable Assertion Checking Engine)
- Generates function level test vectors for reaching target branches



```
function BadMaintainTransitionCost( dz_min::R, dz_max::R, dz_own_ave::R, C_bad_transition::R,
                                    sense_own::Symbol, ra_is_maintain::Bool, ra_is_strengthening::Bool,
                                    s_c::BadTransitionCState )
    R_corrective::R = params().actions.corrective_rate
    R_strengthen::R = params().actions.strengthen_rate
    cost::R = 0.0

    target_branch=false

    if s_c.ra_is_maintain_prev && !ra_is_maintain
        if (s_c.sense_own_prev == :Up) &&
           ( ((dz_min == R_corrective) && (dz_max == Inf)) ||
             ((dz_min == -Inf) && (dz_max == -R_strengthen)) )
            cost = C_bad_transition
        elseif (s_c.sense_own_prev == :Down) &&
               ( ((dz_min == -Inf) && (dz_max == -R_corrective)) ||
                 ((dz_min == R_strengthen) && (dz_max == Inf)) )
            cost = C_bad_transition
        elseif (s_c.sense_own_prev == :Up) && !ra_is_strengthening &&
               ((dz_min == R_strengthen) && (dz_max == Inf))
            target_branch = true
            cost = C_bad_transition
        elseif (s_c.sense_own_prev == :Down) && !ra_is_strengthening &&
               ((dz_min == -Inf) && (dz_max == -R_strengthen))
            cost = C_bad_transition
        end
```

```
    elseif !s_c.ra_is_maintain_prev && ra_is_maintain
        if (sense_own == :Up) &&
           ( ((s_c.dz_min_prev == R_corrective) && (s_c.dz_max_prev == Inf)) ||
             ((s_c.dz_min_prev == R_strengthen) && (s_c.dz_max_prev == Inf)) )
            cost = C_bad_transition
        elseif (sense_own == :Down) &&
               ( ((s_c.dz_min_prev == -Inf) && (s_c.dz_max_prev == -R_corrective)) ||
                 ((s_c.dz_min_prev == -Inf) && (s_c.dz_max_prev == -R_strengthen)) )
            cost = C_bad_transition
        end
        if (abs( dz_own_ave ) < R_corrective)
            cost = cost + C_bad_transition
        end
    end
    assert(target_branch == true)
    return cost::R
end
```

Auxiliary FastPACE branch-tracking code

FastPACE

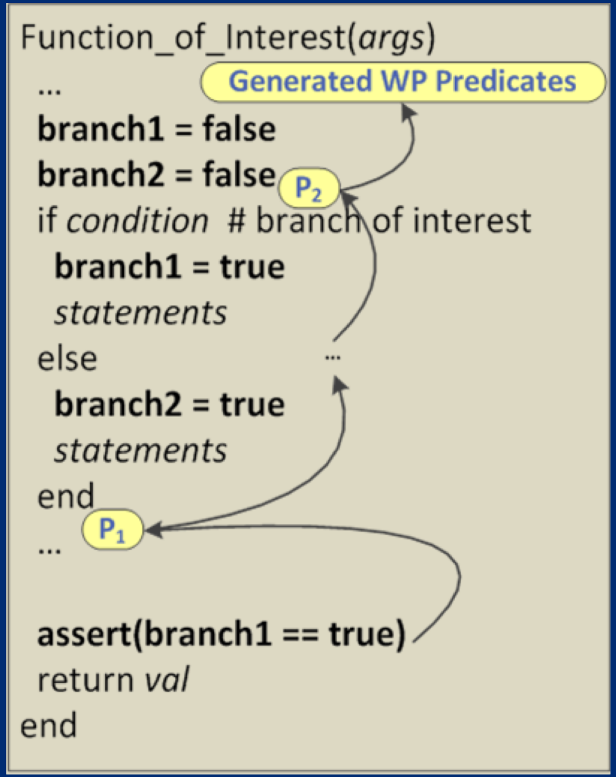
- Allows checking of arbitrary assertions
- Solution can be constrained by adding additional assertions
- Uses SMT solver to compute function inputs satisfying WP
- If assertion is satisfiable returns input parameters to top-level function
- If assertion is unsatisfiable returns *unsat*
- May return *unknown* or timeout

```
BadMaintainTransitionCost(dz_min, dz_max, dz_own_ave,
              C_bad_transition, sense_own, ra_is_maintain,
              ra_is_strengthening, s_c)
```

```
Status: sat
Value: dz_min = 41.6667
Value: dz_max = 9999.0
Value: dz_own_ave = dz_own_ave
Value: C_bad_transition = 0.0
Value: sense_own = sense_own
Value: ra_is_maintain = false
Value: ra_is_strengthening = false
Value: s_c = (BadTransitionCState 3.0 4.0 1.0 true)
```

FastPACE

- Based on Dijkstra’s weakest precondition (WP) analysis with a number of optimizations to improve scalability
  - Single static assignment internal representation
  - Assignments replaced with asserts (Flanagan & Saxe)
  - WP condition simplified for deterministic & non-blocking programs
- Performs constants propagation
- Handles nested compound datatypes, vector and matrix arithmetic
- Loops with variable upper bounds are unrolled to fixed depth, specified by the user
- Inter-procedural analysis handled by in-lining



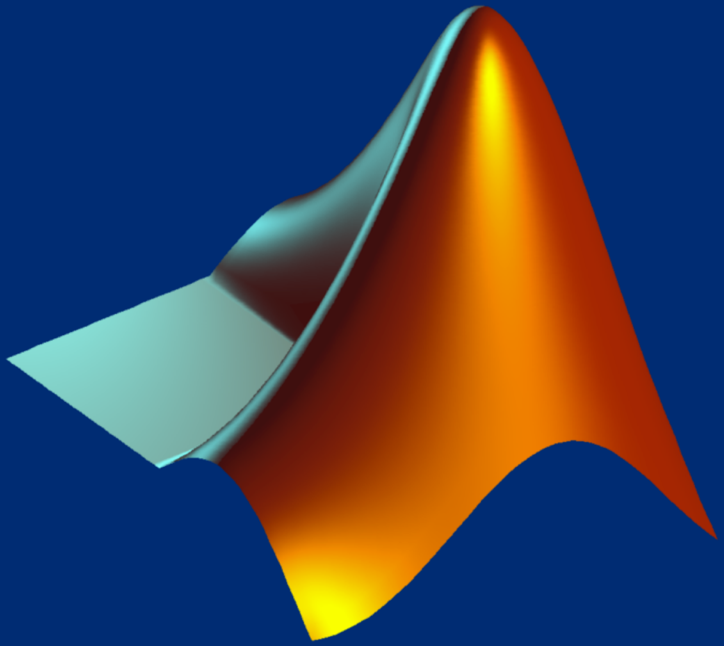
```
Function_of_Interest(args)
  …
  branch1 = false
  branch2 = false
  if condition  # branch of interest
    branch1 = true
    statements
  else
    branch2 = true
    statements
  end
  …

  assert(branch1 == true)
  return val
end
```

FastPACE Architecture

- Preprocessing step (language specific)
  - Julia code is converted into S-expression-like form
  - ACAS X params calls are replaced with constants
  - Identifies user-defined functions and data types
- Weakest precondition generator
  - S-expression code is translated to Guarded Command Language (GCL)
  - Transforms GCL into single static assignment form (Flanagan & Saxe)
  - Creates SMTlib definitions for data types
  - Generates the normal termination expression in SMTlib
- Generated normal termination SMTlib expression is passed to Z3

FastPACE for Matlab

- Initial implementation of Matlab front-end
- Near feature parity with FastPACE Julia (except compound data-types)
- Supports matrices and vectors
- Capable of analyzing simple code



FastPACE on ACAS X

- Over 1300 lines of code analyzed
- Most complex test case -- over 500 LOC
- Identified 22 unreachable branches in DO 385 ADD
  - Many unreachable branches identified in early development versions
- Generated ~20 test cases for DO 385 ADD
  - Numerous test cases developed for early development versions
- <u>Hundreds of man-hours saved</u>

Future work

- Current version is restricted to functionality used in ACAS X
- Add support for dynamic data types, e.g., dictionaries, or strings
- Add standard built-in functions
- Improve nonlinear arithmetic performance

JOHNS HOPKINS

APPLIED PHYSICS LABORATORY

FastPACE

- Front-end is itself written in Julia to take advantage of Julia’s extensive introspection capabilities
  - converts Julia code into S-expression-like like syntax to simplify parsing
  - replaces ACAS X parameter lookups with corresponding constants
  - extracts data types information

ACAS X Architecture

- ACAS X is divided into two effectively independent components
  - Sensor Tracking Module (STM) – receives inputs from onboard sensors, fuses sensor data and provides accurate estimates of the ownship and intruder locations
    - 9727 SLOC
  - Threat Resolution Module (TRM) – receives ownship and intruder locations, tracks potential threats and generates collision resolution advisories when necessary
    - 7709 SLOC
- ACAS X algorithms are written in the Julia language
  - High level language for scientific computation
  - High performance on computation intensive tasks
  - Executable specification