Formally specifying the logic of an automatic guidance controller

David Guaspari

Odyssey Research Associates

Truth arises more readily from error than from confusion.

Francis Bacon
Novum Organum

The Penelope project:

- Interactive, incremental, tool for formal verification of Ada programs (Larch/Ada specifications).
  - Structure or ordinary text editor
  - Permits development of program and proof in concert, "reuse by replay"
- Covers large subset of sequential Ada.
- Mathematically based.

Problem: specify "logic" of experimental Automatic Guidance Control System for a 737

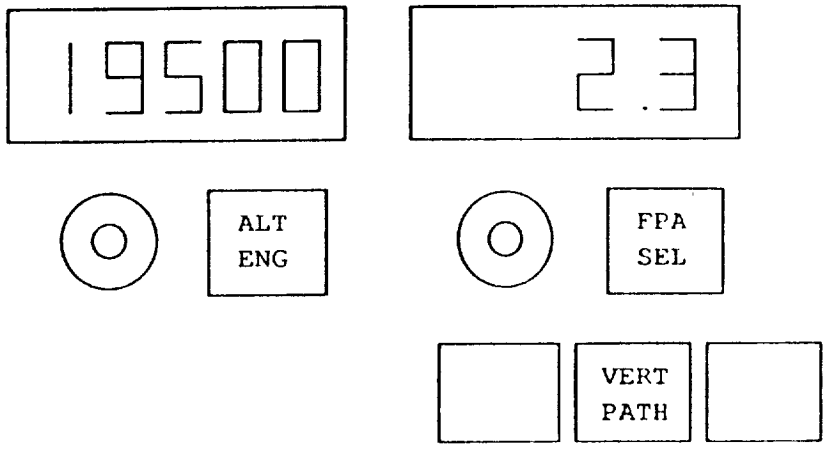
- Pilot requests kind and degrees of automatic assistance
- Requests may be honored, disallowed, "put on hold
- Responses must be displayed

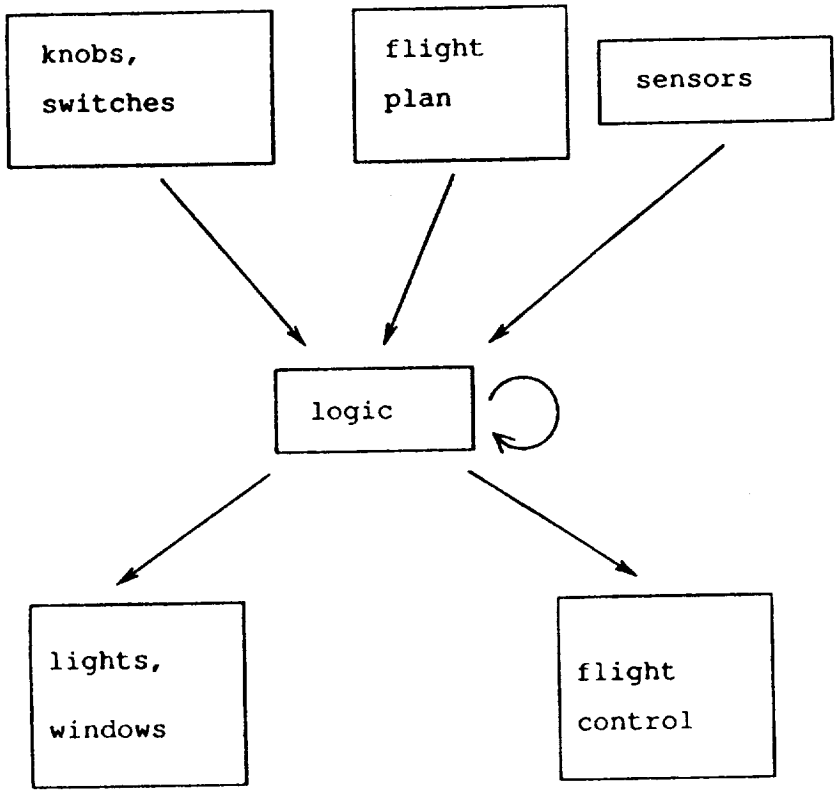
Work-in-progress: Larch/Ada specification

- Formal specification of Ada code
- Goals: precise; intelligible to designers and implementors
- Currently wrong, but clear

Related work

- Original code (CSC)
- Experiment in redesign (NASA)


19500
2.3
ALT
ENG
FPA
SEL
VERT
PATH


knobs,
switches
flight
plan
sensors
logic
lights,
windows
flight
control

Some failures of informal description

1. Ambiguous: "Select" a switch vs. "select" a mode.

2. Incomplete: "CAS ENG may be engaged independent of all other AGCS modes except TIME PATH."

3. Contradictory:

- FPA ... cannot be deselected directly.

- [if] ... appropriate selection of the FPA SEL ... switch returns the mode to the off state ...

Larch/Ada specifications: "two-tiered"

- Mathematical part (Larch Shared Language): defines vocabulary

- Interface part (Larch/Ada): uses vocabulary to specify code

Example: specifying executable addition

Mathematical part: defines mathematical $+$ on *Int*, the (infinite) domain of mathematical integers

Interface part: Specifying evaluation of `x+y`

- Type `integer` is "based on" *Int*.

- Return value $(\mathtt{x} + \mathtt{y})$ if

$$min \leq (\mathtt{x} + \mathtt{y}) \leq max.$$

  No side effects.

- Otherwise, raise `numeric_error`. No side effects.

The mathematical part

States: AGCS_state, Sensor_state, etc.

Actions:

$$\{\text{alt\_eng\_switch},\ldots,\text{alt\_eng\_knob(i)},\ldots,\text{alt\_capture},\ldots\}$$

Modes:

$$\{\text{alt\_eng},\text{fpa\_sel},\text{vert\_path},\ldots\}$$

Transition operation:

$$\text{AGCS\_state, Action, } \ldots \rightarrow \text{AGCS\_state}$$

Observers: active2d, display, ...

Building mathematical part (the AGCS states)

```
AgcsStructure : trait
  AGCS_state record of
    (on: Bool,
    modes: Set_of_modes,
    engaged: Engagement_status,
    setting: Value_settings,
    window: Window_array)
  includes Set(Mode,Set_of_modes)
      ...
  introduces
    transition:
      AGCS_state, Action, Sensor_state,
      Flight_plan → AGCS_state
    initial_on_state: → AGCS_state
  asserts
    ...
```

Description of mode changes caused by switches:

- Is the mode directly deselectable?
- What mode changes result?
- Under what conditions is the mode directly selectable?
- What mode changes result?

Building mathematical part (mode changes)

HorPathSwitch : **trait**
  includes SwitchShell{hor_path}
  asserts for all
    [agcsmodes: Set_of_modes,
    pl: Flight_plan,
    sens: Sensor_state]

    hor_path_deselectable
    hor_path_selectable(agcsmodes,pl) =
      (auto $\in$ agcsmodes) $\wedge$ active2d(pl)
    hor_path_selection_result(agcsmodes,sens,pl) =
      [hor_path] $\cup$ [[cas]]
    hor_path_deselection_result(agcsmodes) =
      [tka_sel] $\cup$ [[cas]]

Intuitive description of window status (*chosen* vs. *current*):

- The w_knob makes the corresponding w-window chosen.

- Any action selecting the w mode makes the w-window chosen.

- Any action deselecting the w mode makes the w-window current.

- Any other action leaves the status of the w-window unchanged.

Building the mathematical part (window changes)

StatusShell : **trait**
  imports AgcsStructure
  introduces
    #.component :
      Window_array → Window_status
    md: → Mode
    knob : Value → Action
  asserts for all [agcs:AGCS_state, ...]
  abbreviation
    agcs' == transition(agcs,act,sensor,plan)
    agcs'.window.component =
      if md ∈ agcs'.modes - acgs.modes
        then chosen
      elsif md ∈ agcs.mode − agcs'.modes
        then current
      elsif act = knob(i) **then** chosen
      else agcs.window.component

Example: StatusShell{alt,alt_eng,Airspeed}

C-6.

Design of the code:

- Packages `panel_logic`, `display_manager`, `sensor_data`, `flight_plan`, `flight_control`.

- State of `panel_logic` based on AGCS_state, etc.

- Actions $\mapsto$ procedures of `panel_logic`:
  - read state of `panel_logic`, `sensor_data`, `flight_plan`
  - modify states of `panel_logic`, `display_manager`, `flight_control`

- Consistent with polling, interrupts, etc.

Specifying the code:

```
--| WITH TRAIT AgcsLogic, AgcsProperties,
--|             LogicalDisplay
--| WITH sensor_data, flight_plan,
--|      display_manager, flight_control

with sensor_data_types; use sensor_data_types;
package panel_logic
   --| BASED ON AGCS_state
   --| INVARIANT
   --|  panel_logic.on -> good(panel_logic)
   --| INITIALLY not panel_logic.on
            ...

 end panel_logic;
```

```
procedure att_cws_switch;
--| WHERE
--|  GLOBALS IN  panel_logic
--|  GLOBALS OUT display_manager,
--|              flight_control,
--|              panel_logic

--|  IN panel_logic.on

--|  OUT panel_logic =
--|   transition(IN panel_logic,
--|              att_cws_switch,*,*)
--|  OUT FORALL ss: Sensor_state::
--|   look(display_manager,ss) =
--|   display(panel_logic,ss)
--|  OUT FORALL md:mode ::
--|   fc_engaged(md,flight_control) =
--|   engaged(md,panel_logic)
--| END WHERE;
```

```
procedure turn_on_agcs
--| WHERE
          ...
--|    OUT panel_logic = initial_on_state
          ...
--| END WHERE;
```