Using SPARK to ensure System to Software Integrity

Tonu Naks, M. Anthony Aiello, S. Tucker Taft

AdaCore

10th European Congress on Embedded Real Time Software and Systems, 29-31/01/2020 Toulouse

Agenda

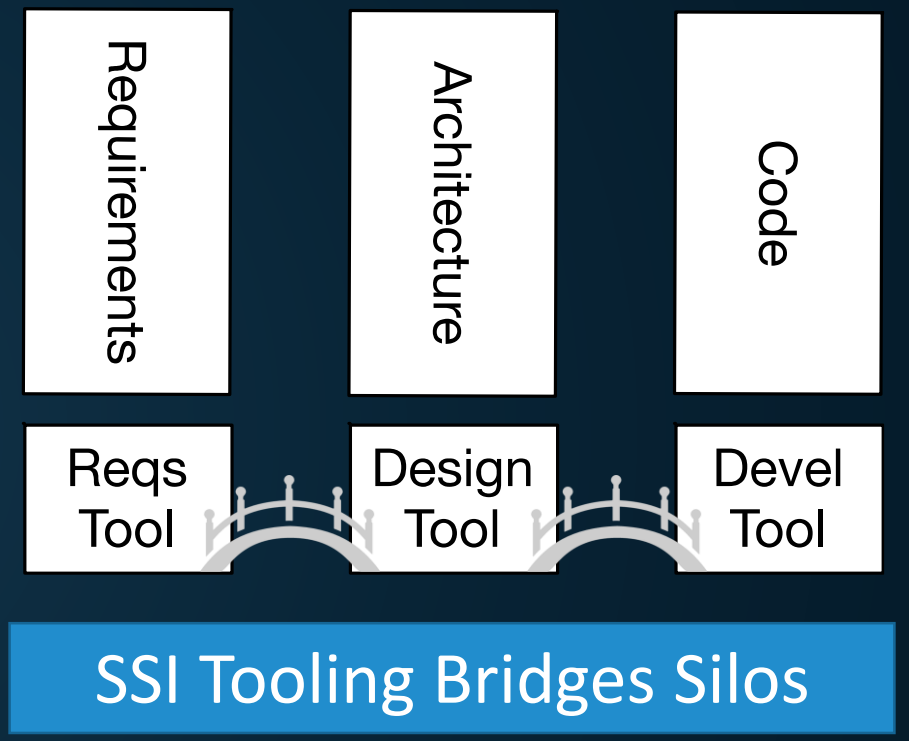
- AdaCore System-to-Software Integrity (SSI) initiative
- Workflow in a nutshell
- Workflow demonstrated by a case study
- Challenges/open questions/next steps

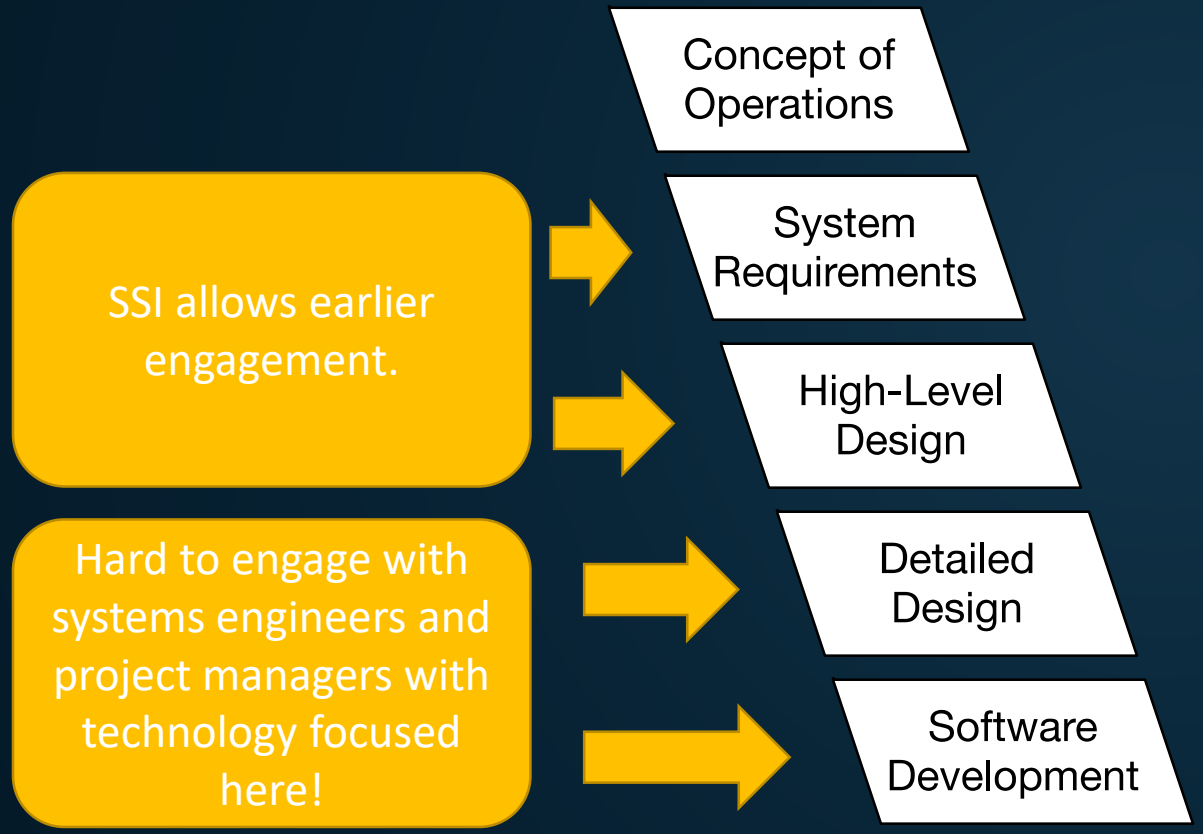
SSI

System-to-software integrity is a desired trait of high-assurance systems engineering.

- Ensure development process yields adequate assurance
- Link artifacts at different levels with formal properties & tool support
- Help engineers in moving from level to level with smart translations
- Reduce information loss in communication of various teams

System-level properties maintained through each development step until realized in software.



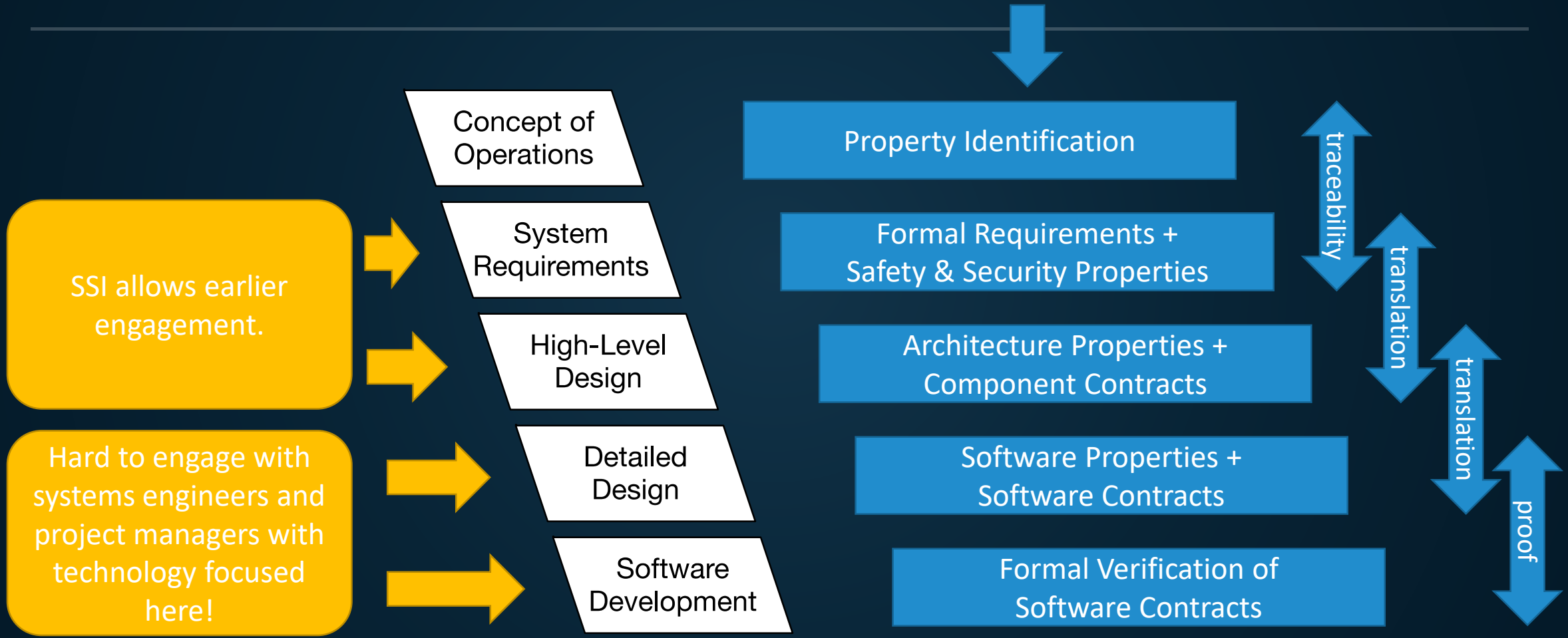


SSI

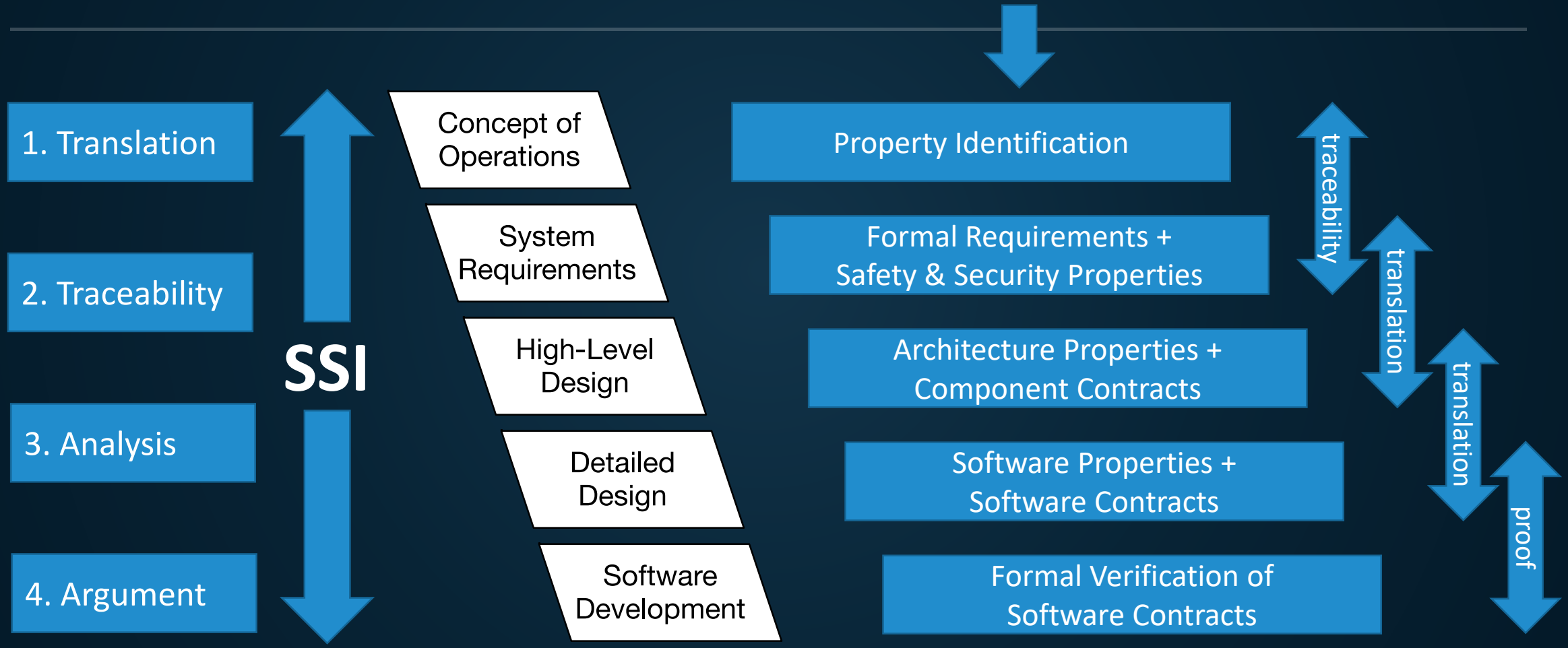
System-level properties maintained in software

- Hard for software engineers to identify application-specific properties
- Hard for systems engineers to think about software-level properties
- SSI allows early engagement and property continuity

SSI


Concept of Operations
System Requirements
High-Level Design
Detailed Design
Software Development
SSI allows earlier engagement.
Hard to engage with systems engineers and project managers with technology focused here!
Property Identification
Formal Requirements + Safety & Security Properties
Architecture Properties + Component Contracts
Software Properties + Software Contracts
Formal Verification of Software Contracts
traceability
translation
translation
proof

SSI


1. Translation
2. Traceability
3. Analysis
4. Argument
SSI
Concept of Operations
System Requirements
High-Level Design
Detailed Design
Software Development
Property Identification
Formal Requirements + Safety & Security Properties
Architecture Properties + Component Contracts
Software Properties + Software Contracts
Formal Verification of Software Contracts
traceability
translation
translation
proof

SSI

System-level properties maintained in software

1. Translation

- Translate Properties from one “level” to the next
- Example: properties for requirements -> properties as contracts in a design.
- Property decomposition may be required

2. Traceability

- Bidirectional traceability of properties across “levels”
- Trace properties to models & code
- Monitor for broken links

3. Analysis

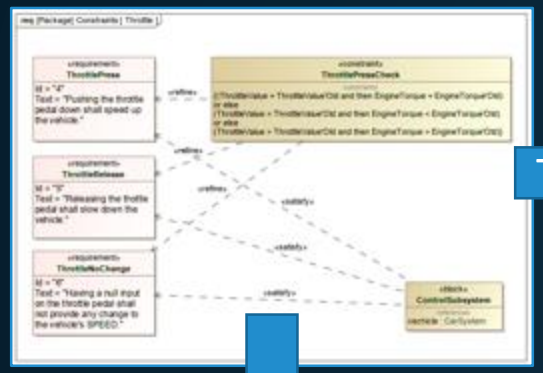
- Vertical: prove that properties are consistent across levels
- Horizontal: prove that decompositions satisfy higher-level properties

4. Argument

- SSI evidence may need logical induction to justify fully
- Present & justify evidence where deduction is not fully possible
- Provide support for certification

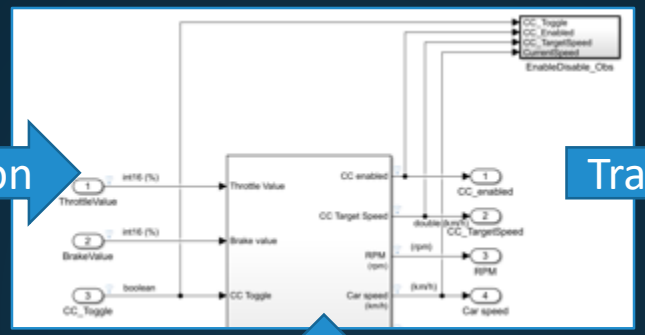
SSI tooling example

SysML Requirements Diagram



Translation

Simulink Synchronous Observer



Translation

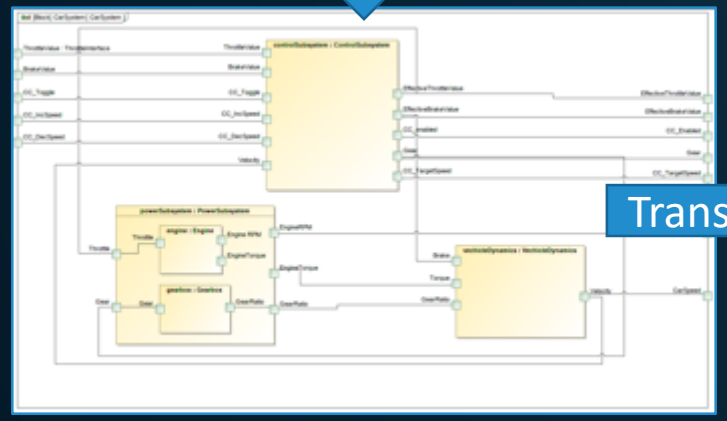
SPARK Contracts

```
package car_system is

   procedure initStates (State : in out car_system_State);

   procedure initOutputs (State : in out car_system_State);

   procedure comp
     (Throttle_Value : Integer_16_m0_0_M100_0;
      Brake_value : Integer_16_m0_0_M100_0;
      Cruise_Control : Boolean;
      CC_Speed : Boolean;
      Resec_Speed : Boolean;
      CC_enabled : out Boolean;
      CC_Target_Speed : out Long_Float_m30_0_M100_0;
      RPM : out Long_Float;
      Car_speed : out Long_Float;
      Gear : out Integer_8;
      Throttle_input : out Integer_16_m0_0_M100_0;
      Brake_input : out Integer_16_m0_0_M100_0;
      State : in out car_system_State);

   procedure up (State : in out car_system_State);
```

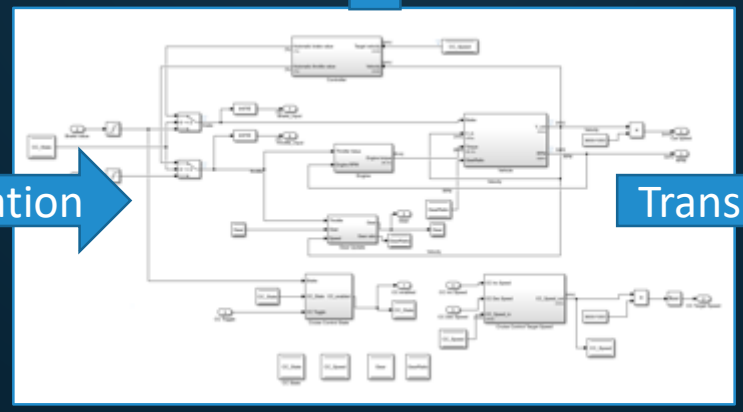
Manual Refinement

QGen Verifier

GNATProve



Translation



Translation

```
package body car_system is

   procedure initStates (State : in out car_system_State) is
   begin
      --  Start of function
      --  Block 'car_system/Car System/Physical model/Vehicle/Discrete-Time Integrator'
      State.Discrete_Time_Integrator_in_memory := 0.0E+00;
      State.Discrete_Time_Integrator_out_memory := (qgen_base_workspace.V_init) / (3.6);
      --  End Block 'car_system/Car System/Physical model/Vehicle/Discrete-Time Integrator'

      --  Block 'car_system/Car System/Controller/If1'
      Controlled_input.initStates (State.Controlled_input_memory);
      --  End Block 'car_system/Car System/Controller/If1'

      --  Block 'car_system/Car System/Controller/Merge throttle'
      State.Merge_throttle_out1 := 0;
      --  End Block 'car_system/Car System/Controller/Merge throttle'

      --  Block 'car_system/Car System/Controller/Merge brake'
      State.Merge_brake_out1 := 0;
      --  End Block 'car_system/Car System/Controller/Merge brake'

      --  Block 'car_system/Car System/Cruise Control Target Speed/Merge'
      State.Merge_out1 := 0.0;
      --  End Block 'car_system/Car System/Cruise Control Target Speed/Merge'

      --  Block 'car_system/Car System/CC State'
      State.CC_State := False;
```

SysML Internal Block Diagram

Simulink Subsystem

SPARK Code

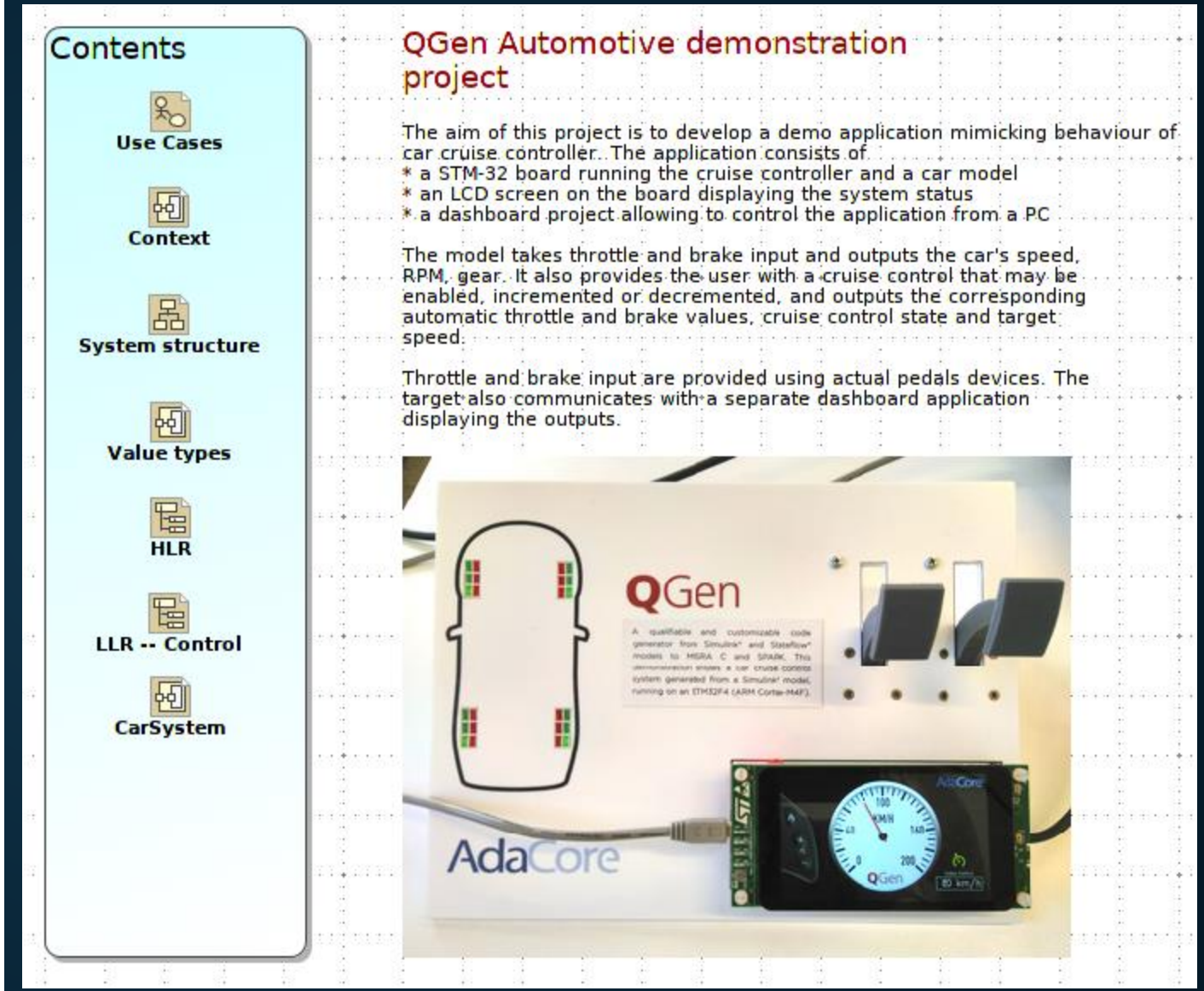
A Case Study

Autopilot Simulator

SysML → Simulink → SPARK

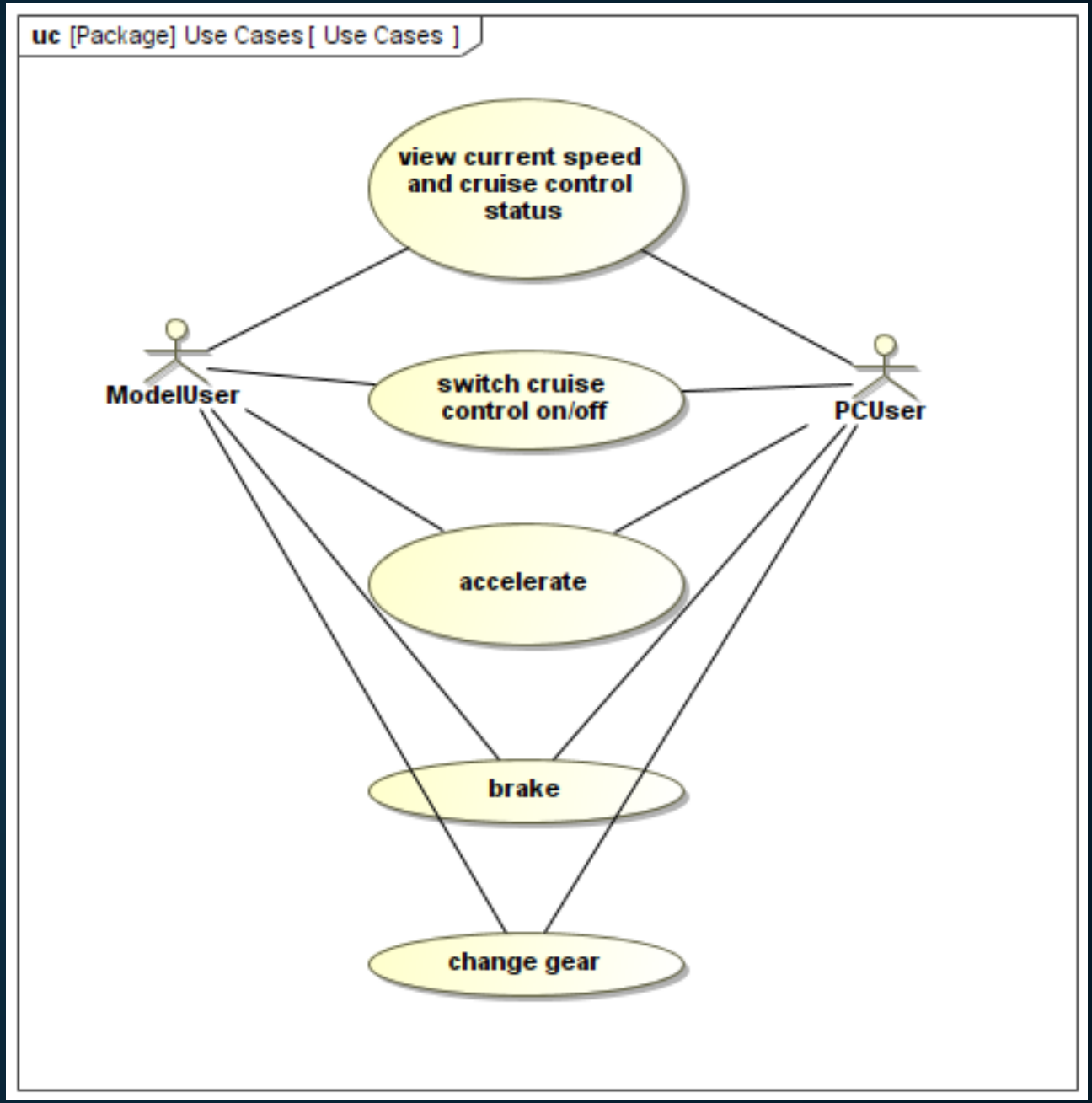
A sample workflow

- A simple demo application mimicking behavior of a car cruise controller
  - STM-32 board running the cruise control and car model
  - LCD screen on the board emulating car cockpit displays
  - A dashboard application allowing to control the board from PC



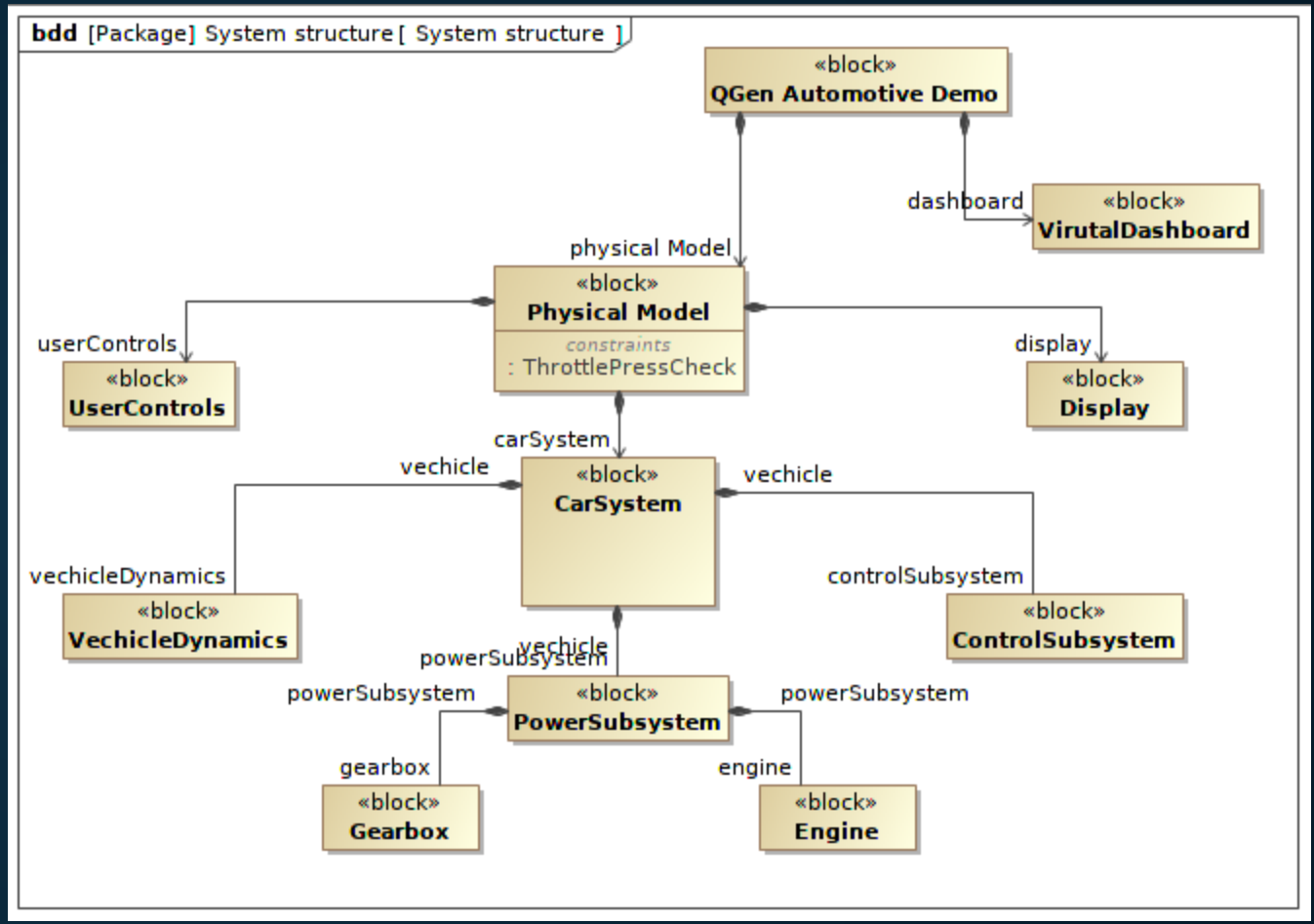
Stakeholders & Use-Cases

- Two types of users:
  1. **ModelUser**: manipulates the system through physical controls on the demo box
  2. **PCUser**: controls the system through PC application
- Both have access to the same use-cases



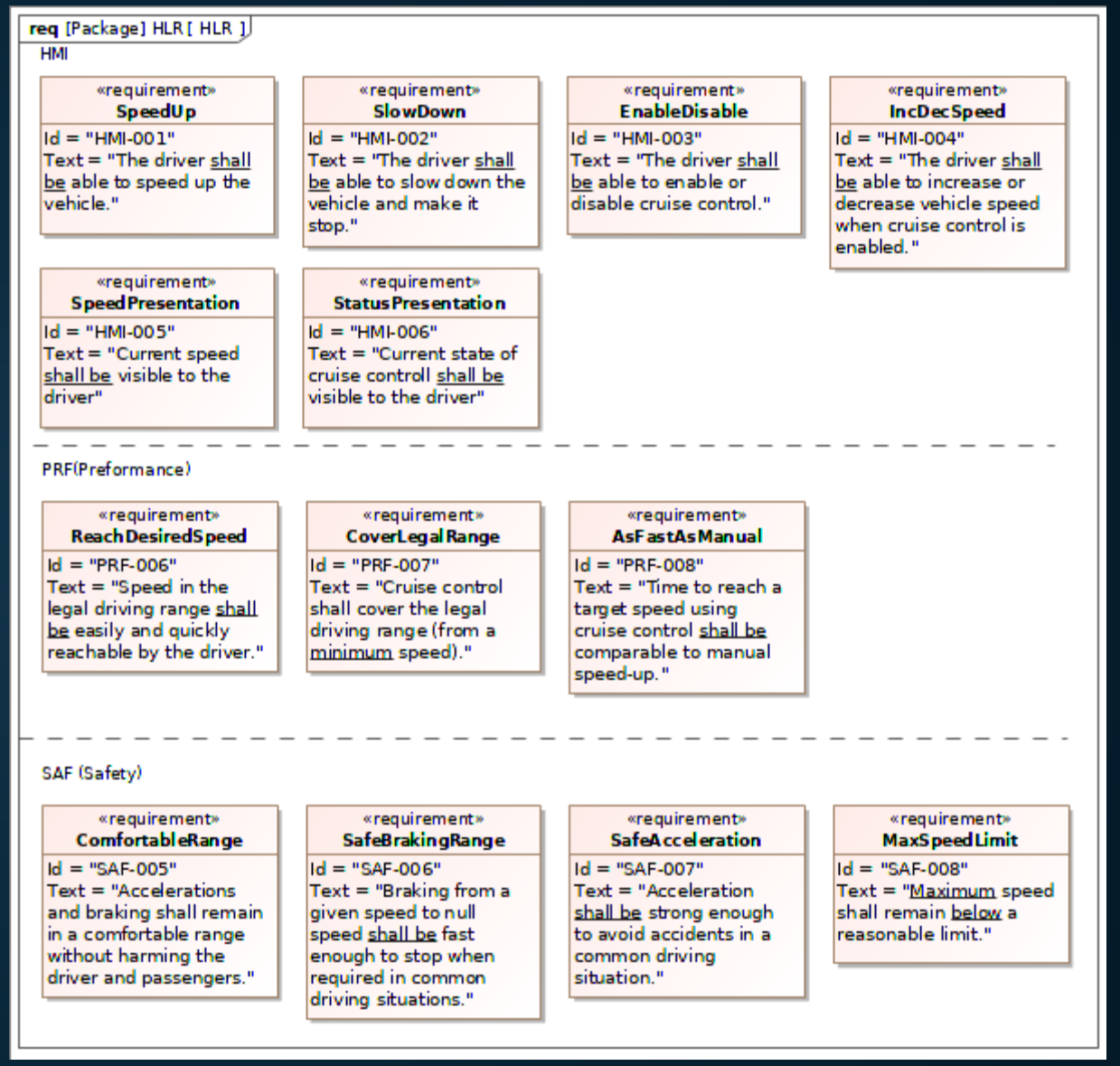
Main Components

- System divided into two main parts:
  1. **PhysicalModel**: implements
     - vehicle simulation and
     - cruise control
  2. **VirtualDasboard**: allows access from PC



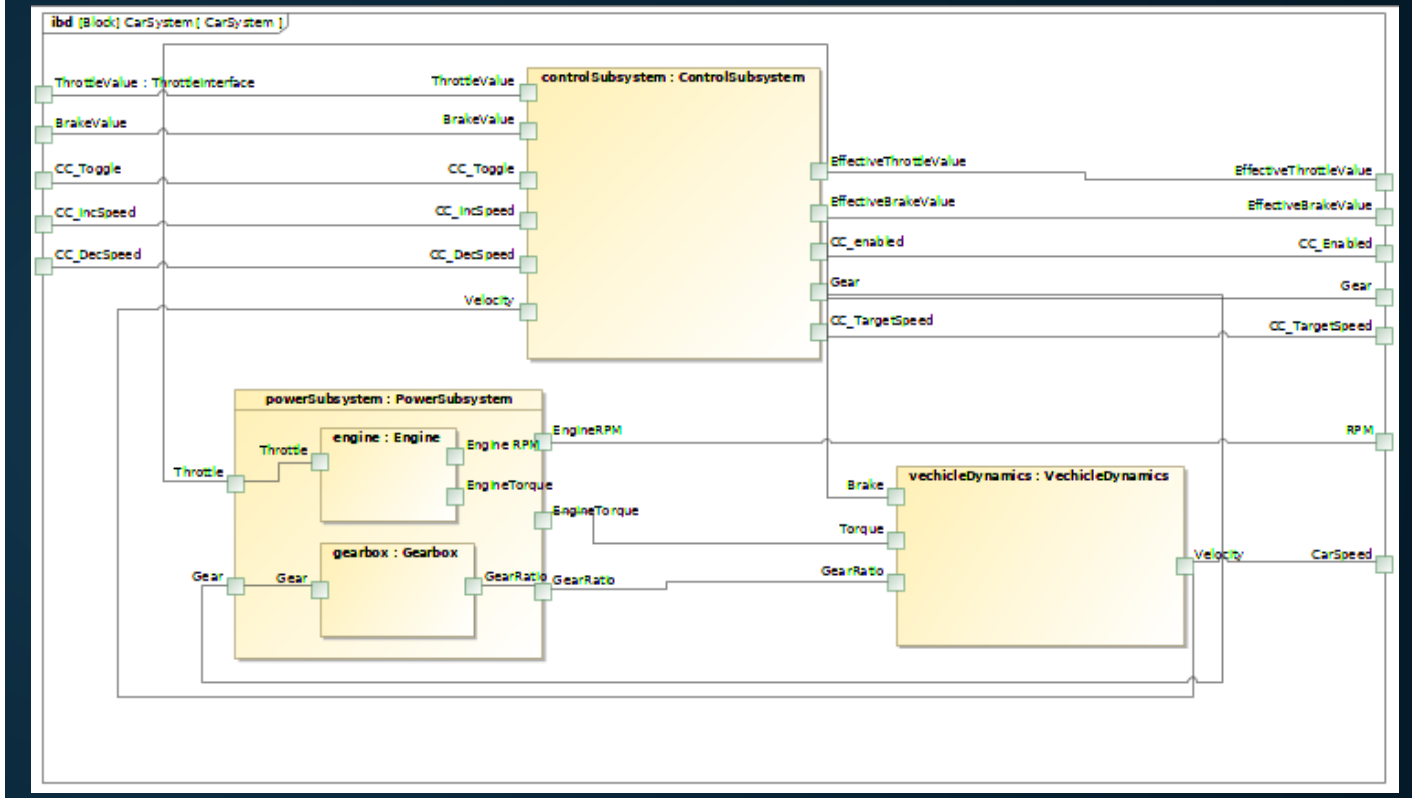
High-level requirements

- Initially, requirements are defined textually
- Formalization of selected subset apply to
  - High-Level Requirements or
  - Low-Level Requirements



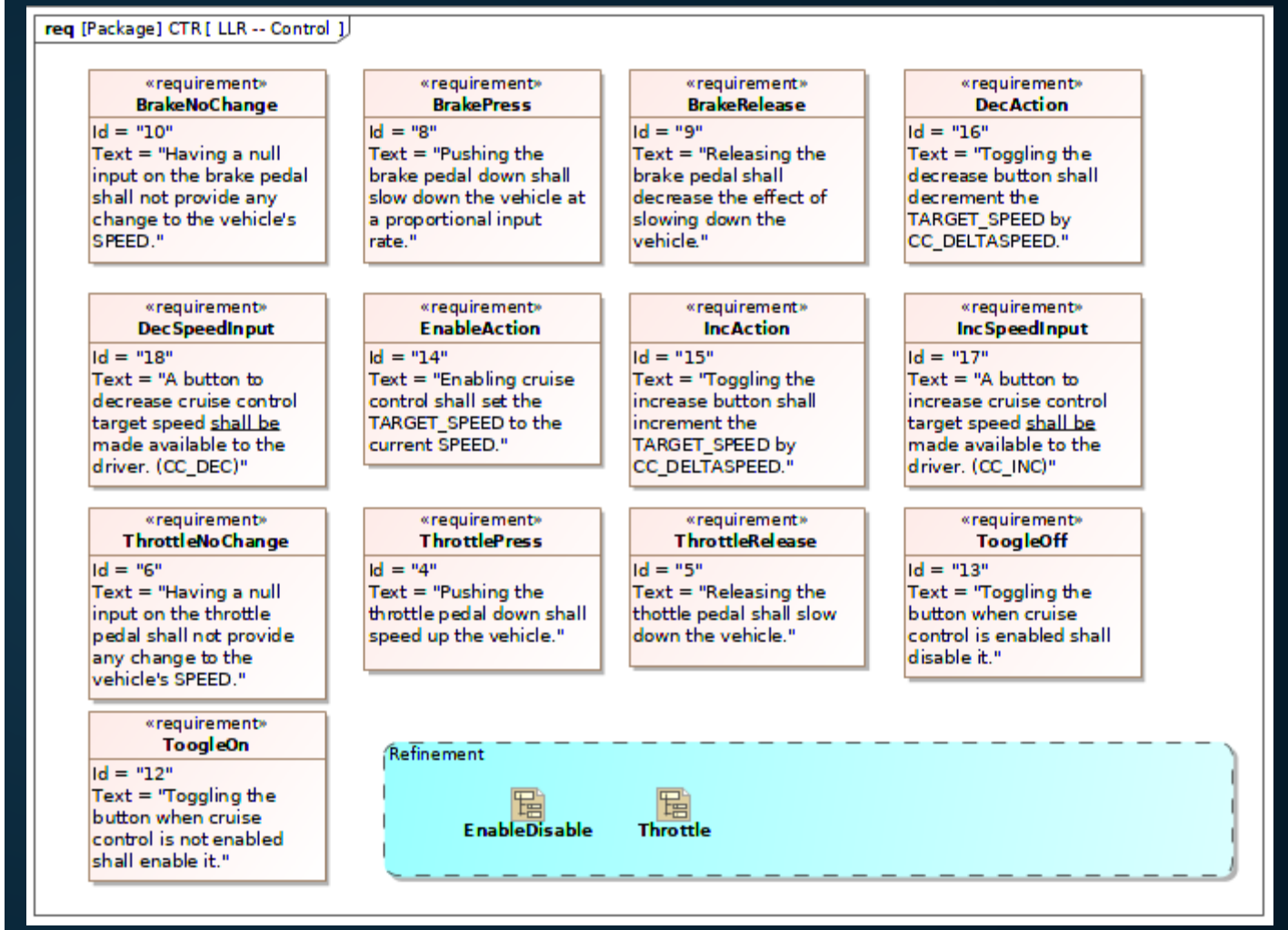
Internal Structure & Data Flows

- Defining the internal structure provides high-level division into software components
- Interface definitions provide names and types for further decomposition and formalization of the requirements



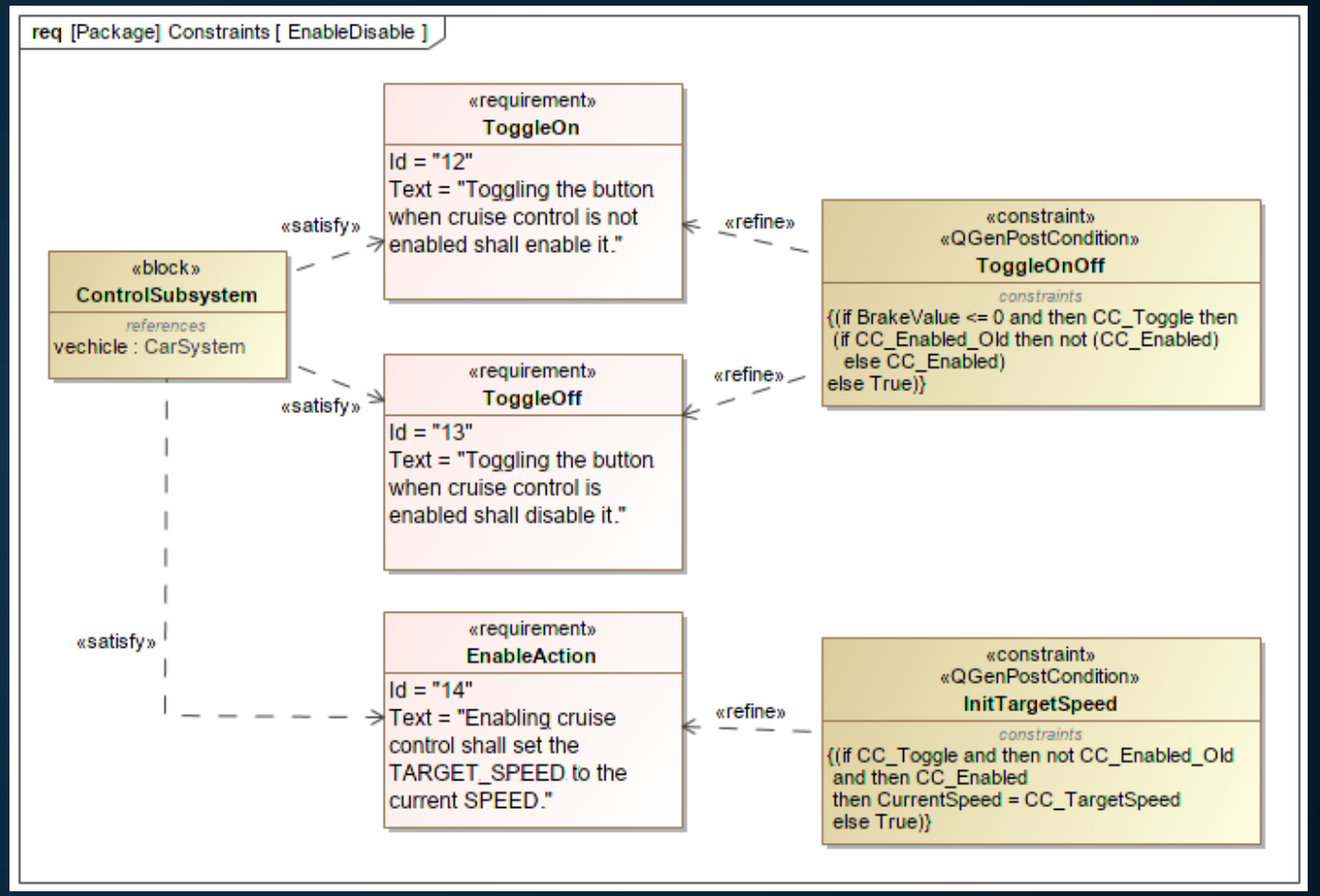
Low-Level Requirements

- Redefine the High-Level Requirements
- Use interface names defined in system high-level architecture
- Specify functional behavior for each component



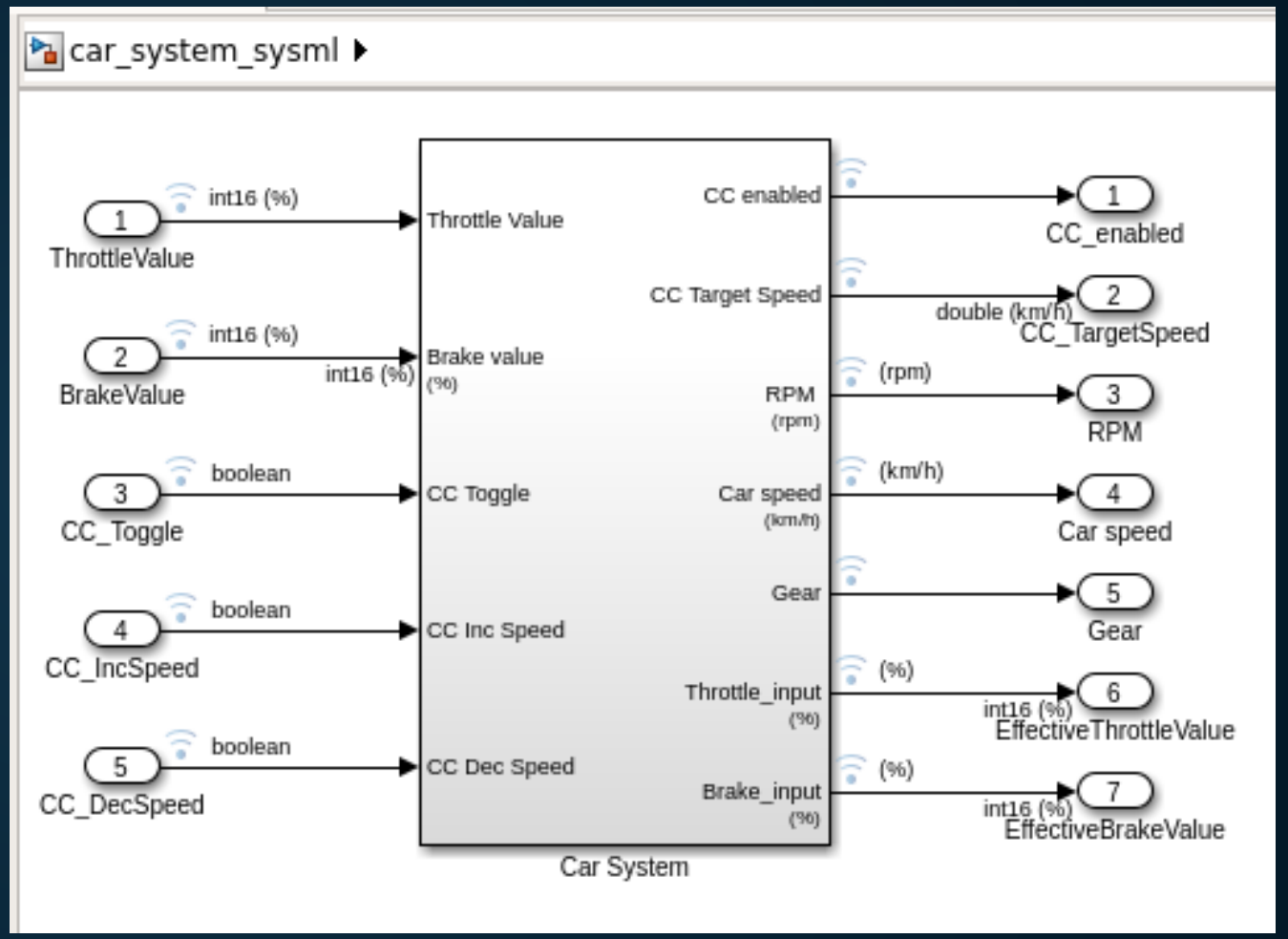
Requirement Formalization

- Rewrite requirements as constraints
  - allows consistency checks between requirements, design, and implementation
- Allocate requirements to components
- Here, we have chosen SPARK as the language for formalization



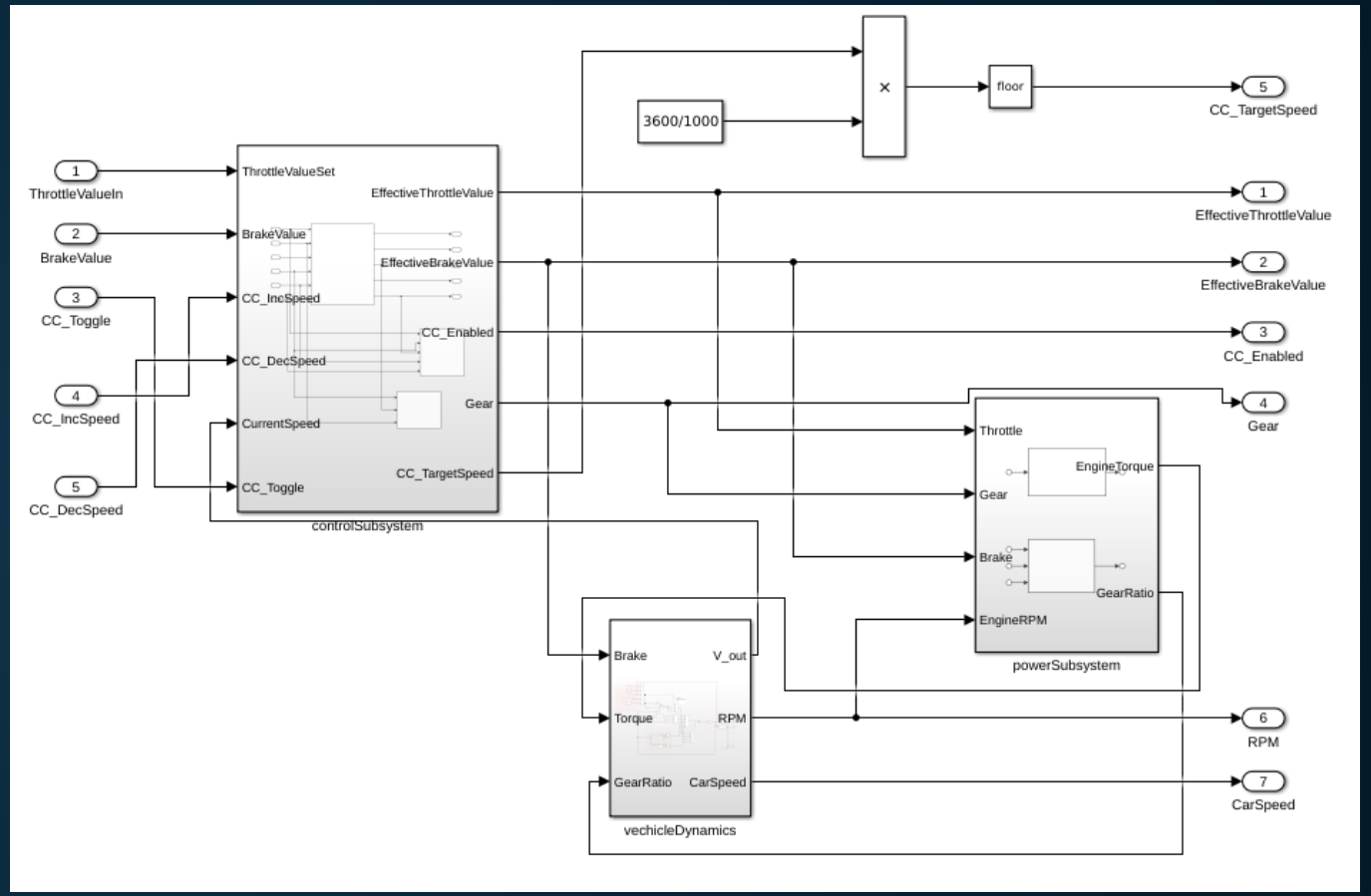
Conversion to Simulink

- Aim of Simulink conversion:
  - provide a skeleton for refining the design by defining computation algorithms
  - validate the system definition by simulation



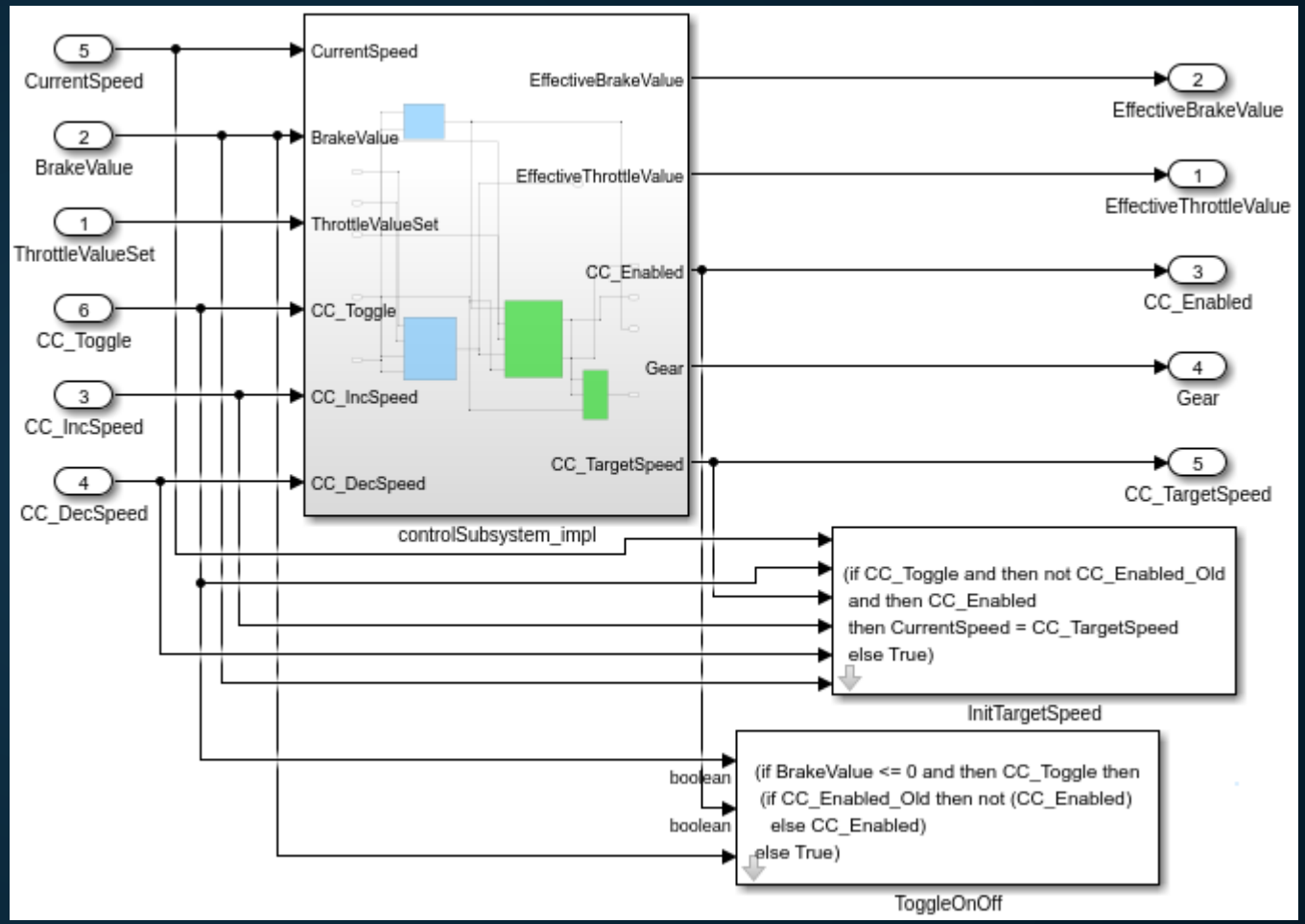
Internal Structure → Simulink

- Convert blocks from IBD to Simulink
- Provide skeletons / containers for
  - control algorithms
  - plant model



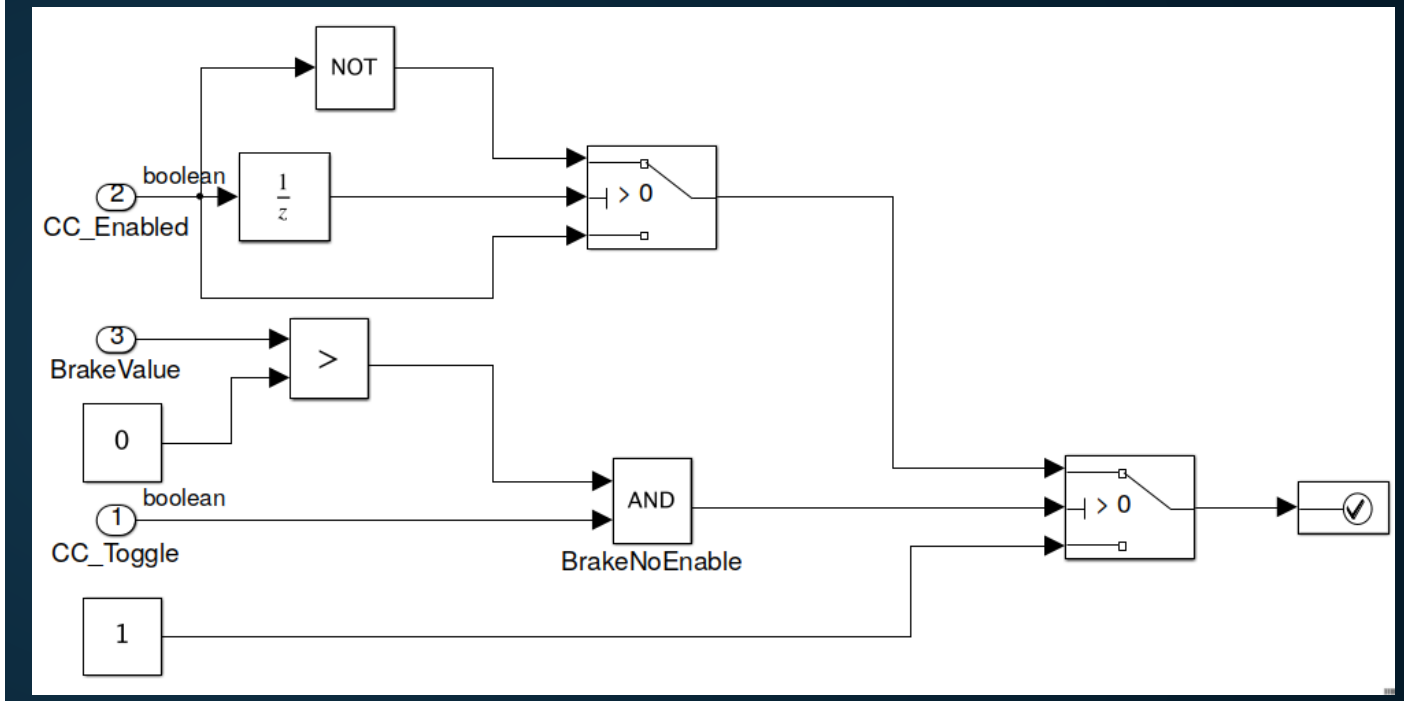
Requirements → Simulink Observers

- The requirements formalized by constraints are inserted in Simulink as synchronous observers
- Block mask tells the code generator that subsystem contents should be handled as a post-condition



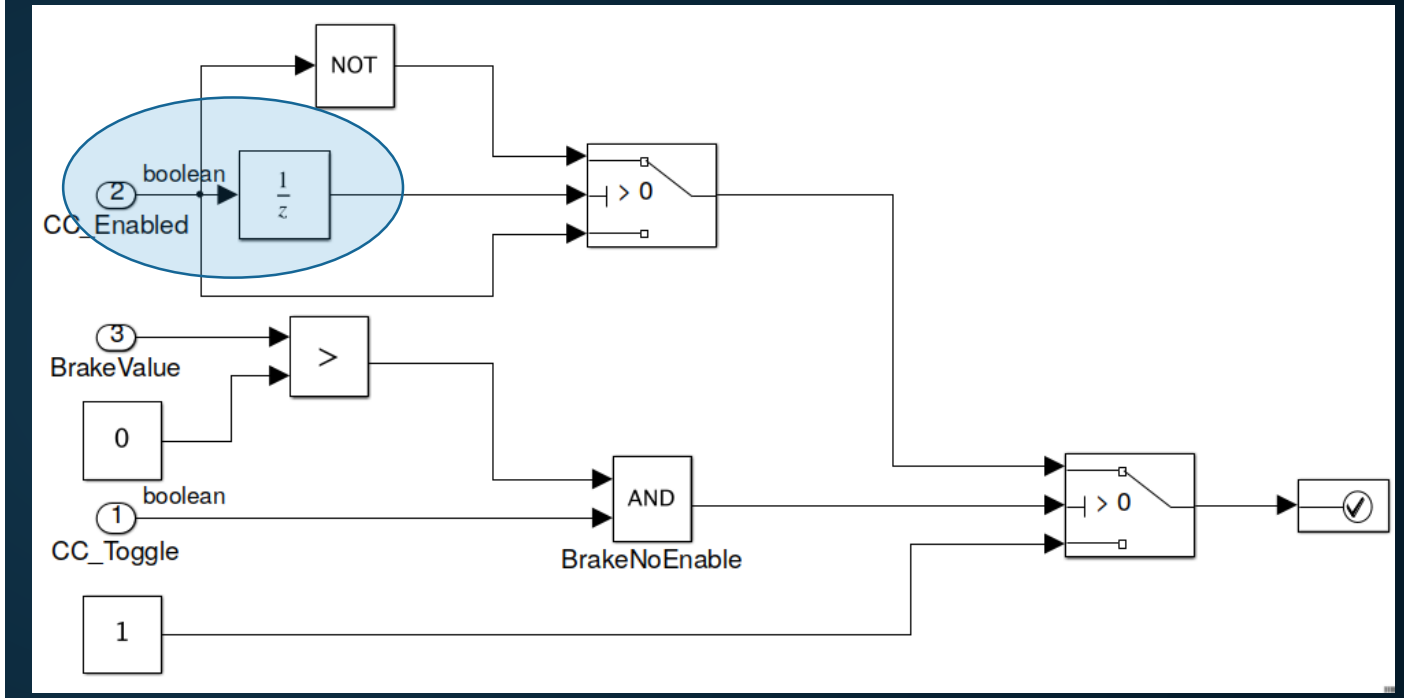
Observer Contents

- A QGen observer is a subsystem that
  - takes signals from functional part of the model as input
  - compares signal values with
    - each other
    - an oracle defined by constraints in SysML
  - raises an exception when comparison fails



Reasoning About Time

- A simplified way of inserting the time in constraints is to refer to previous computation steps
- Here the modeler has a choice to either
  - insert the memory buffer explicitly and refer to this
  - rely on ‘Old mechanism in Ada
- To mimic the ‘Old behavior in Simulink we use the UnitDelay block



Contract in generated code

- Each observer block is converted to a check function

```
package ToggleOnOff is

   function check
     (CC_Toggle : Boolean;
      CC_Enabled : Boolean;
      BrakeValue : Integer_16;
      CC_Enabled_Old : Boolean)
       return Boolean
   is (if BrakeValue > 0 and then CC_Toggle then
         (if CC_Enabled_Old then not CC_Enabled
          else CC_Enabled)
       else True);

end ToggleOnOff;
```

Contract in generated code

- The check function is called from pre- or postcondition of a functional subsystem
- Internal memory blocks in observers are replaced with ‘Old actuals

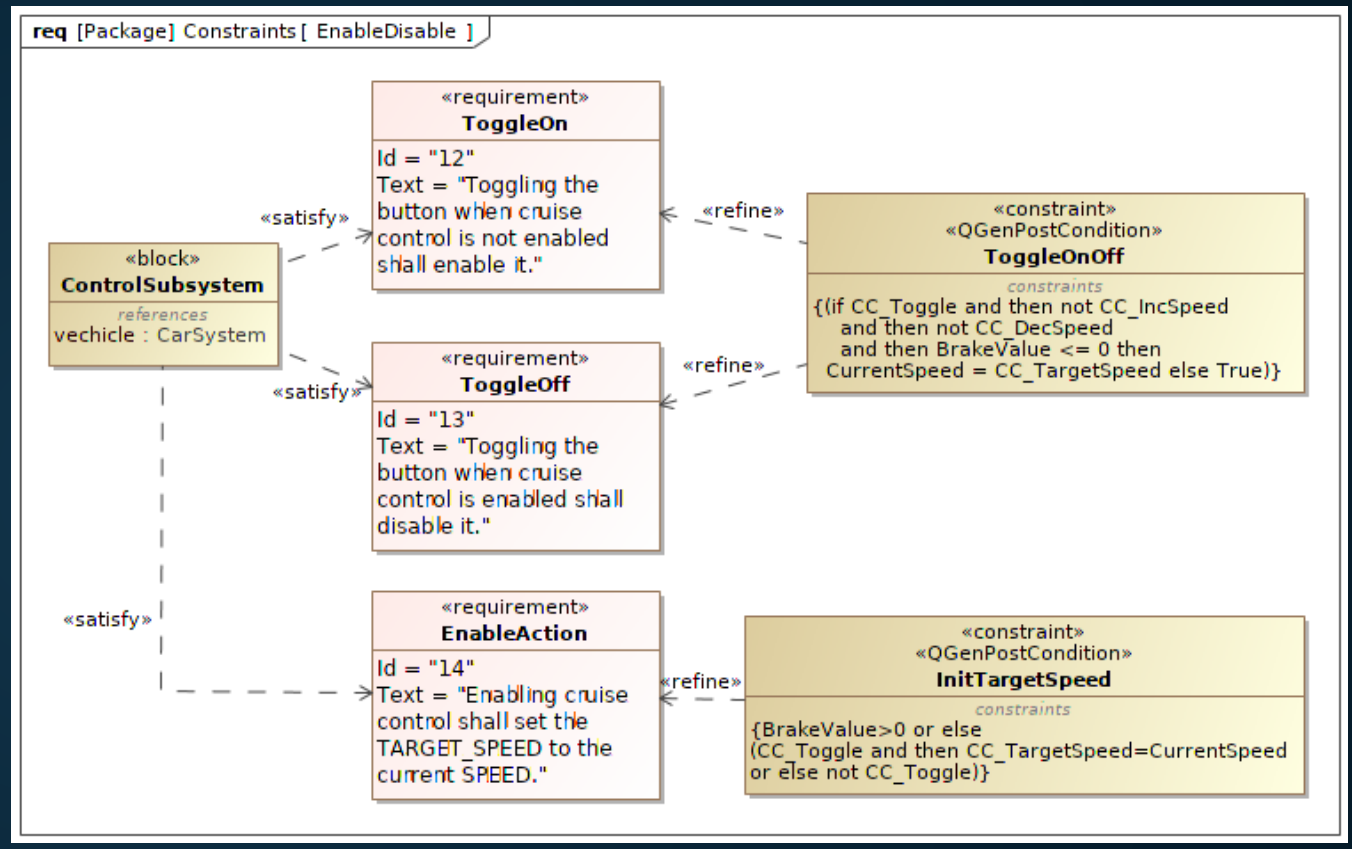
```
package controlSubsystem is

   procedure initStates (State : in out controlSubsystem_State);

   procedure initOutputs (State : in out controlSubsystem_State);

   procedure comp
      (ThrottleValueSet : Integer_16;
       BrakeValue : Integer_16;
       CC_IncSpeed : Boolean;
       CC_DecSpeed : Boolean;
       CurrentSpeed : Long_Float;
       CC_Toggle : Boolean;
       EffectiveThrottleValue : out Integer_16;
       EffectiveBrakeValue : out Integer_16;
       CC_Enabled : out Boolean;
       Gear : out Integer_8;
       CC_TargetSpeed : out Long_Float;
       State : in out controlSubsystem_State)
   with
   Post =>
         (ToggleOnOff.check
            (CC_Toggle, CC_Enabled, BrakeValue,
             CC_Enabled'Old))
         and
           (InitTargetSpeed.check
              (CurrentSpeed, CC_Toggle, CC_TargetSpeed,
               CC_IncSpeed, CC_DecSpeed, BrakeValue));

   procedure up (State : in out controlSubsystem_State);

end controlSubsystem;
```

Formalizing requirements

- Parametric diagrams
  - Good for physical phenomena – the „plant model“
  - May need „creative interpretation“ while translating to software constraints
- Activity diagrams/state models
  - Potential candidates for draft algorithm design or test oracle
  - Equivalence proofs not trivial (if possible at all) after refinements in subsequent design steps
- **Constraint blocks**
  - Good form for representing axiomatic definitions of properties and their relationships
  - Easy to carry forward to the next levels and backpropagate changes

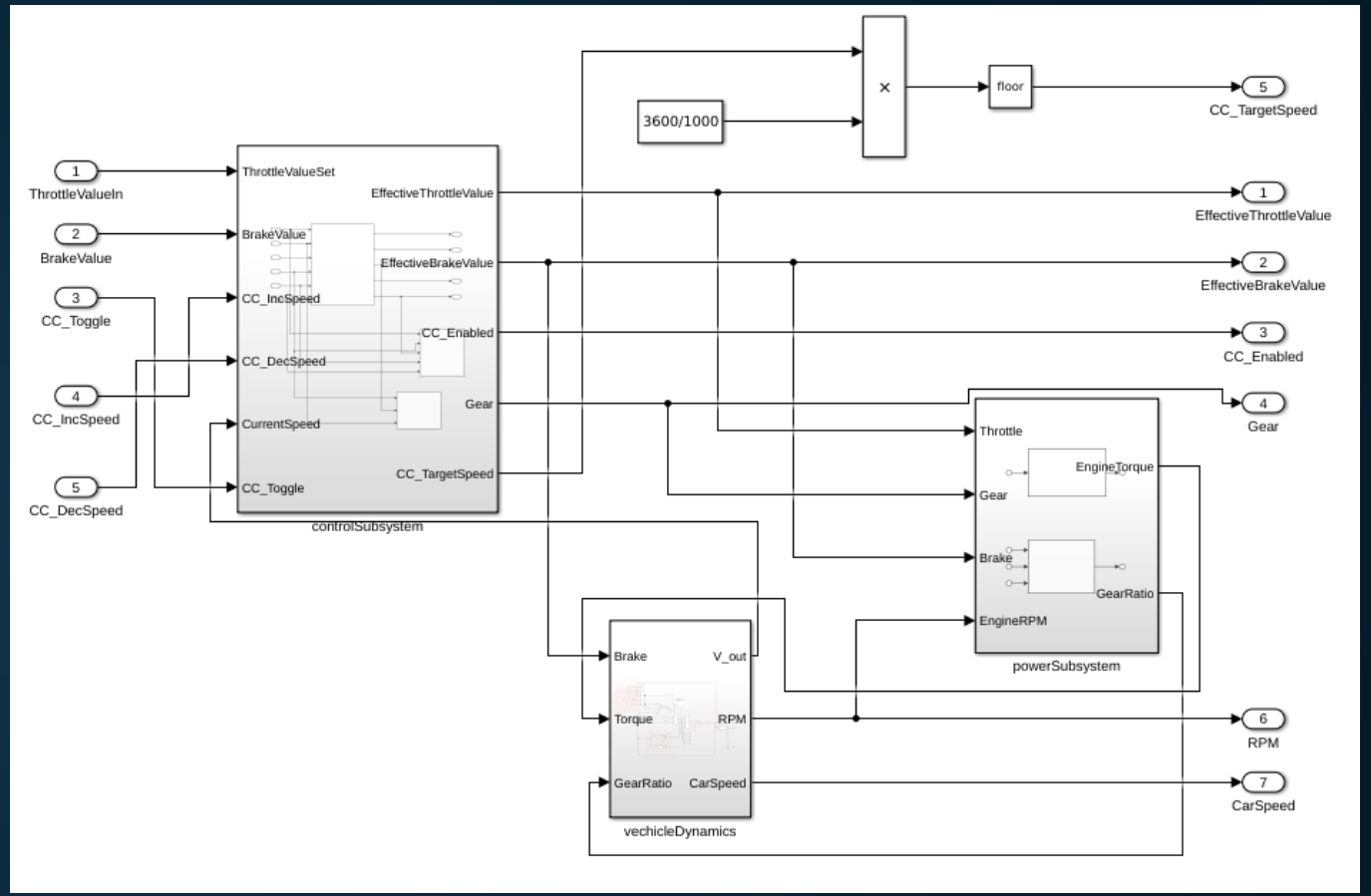
Why SPARK in SysML?

- Looking for axiomatic specifications potentially with late binding
- OCL seems too strictly defined for this purpose (e.g. pre and postconditions bound to behaviors) => using a different language rather than loosening the constraints
- The current converter is easily extensible to support OCL or some other expression language



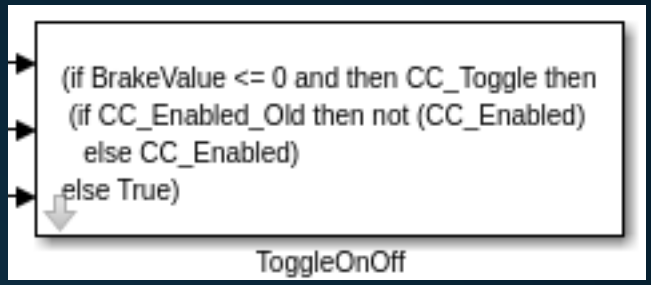
The Role of Simulink

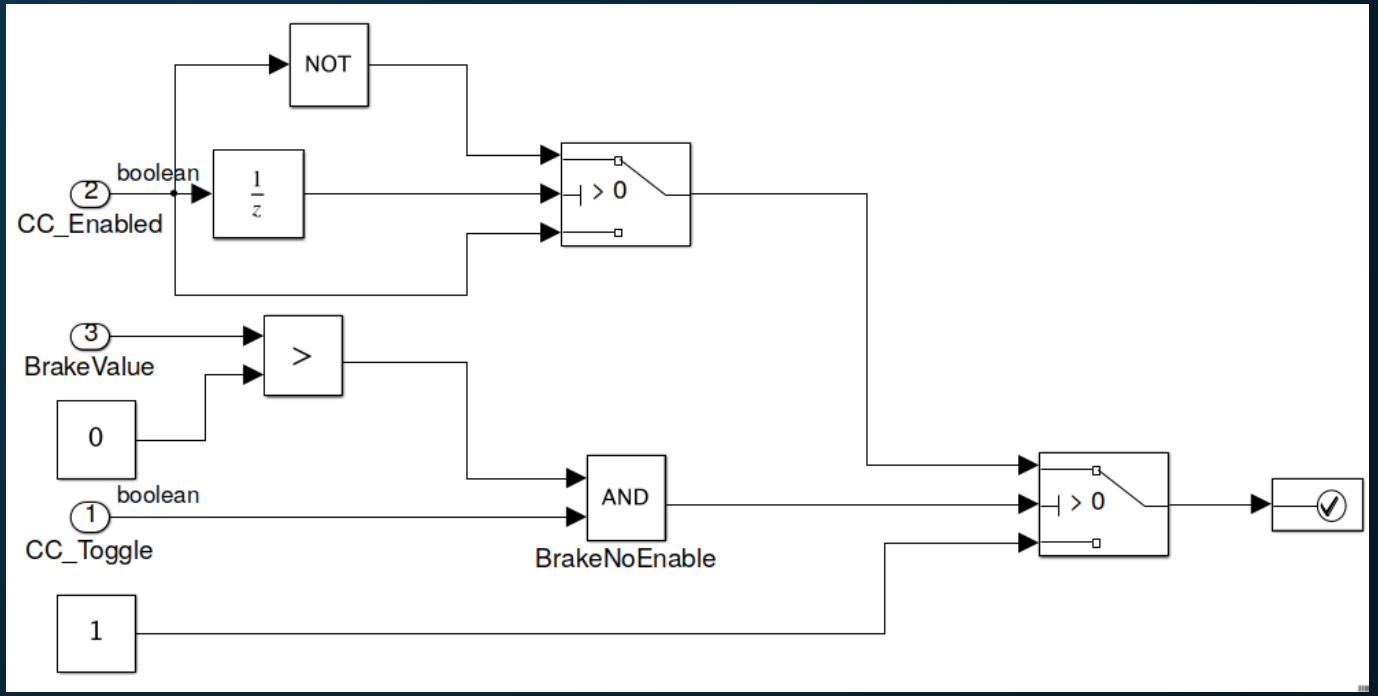
- An appropriate tool for algorithm design
- More natural choice for a control engineer than activity or parametric diagrams
- Qualifiable automated workflow from Simulink to code already exists (QGen)



Observers in Simulink

- SPARK expression would be sufficient for code generation and simulation (using a s-function)
- Difficult to validate and modify in Simulink
- Block diagram simplifies contract refinement at simulation time





Questions/challenges/next steps

- Relation between parametric diagrams and constraints?
- Good workflow for binding the constraint expression with block properties?
- Composability and validation of the constraints
  - First formalization in SysML where the only validation mechanism is review
  - Easy to validate in Simulink or source code but this is too late for systems engineer
  - Achieving completeness assumes iterations between system design and algorithm design
- Support for automatic proof
  - Need for additional hints about code to successfully prove postconditions

Thank you!

AdaCore