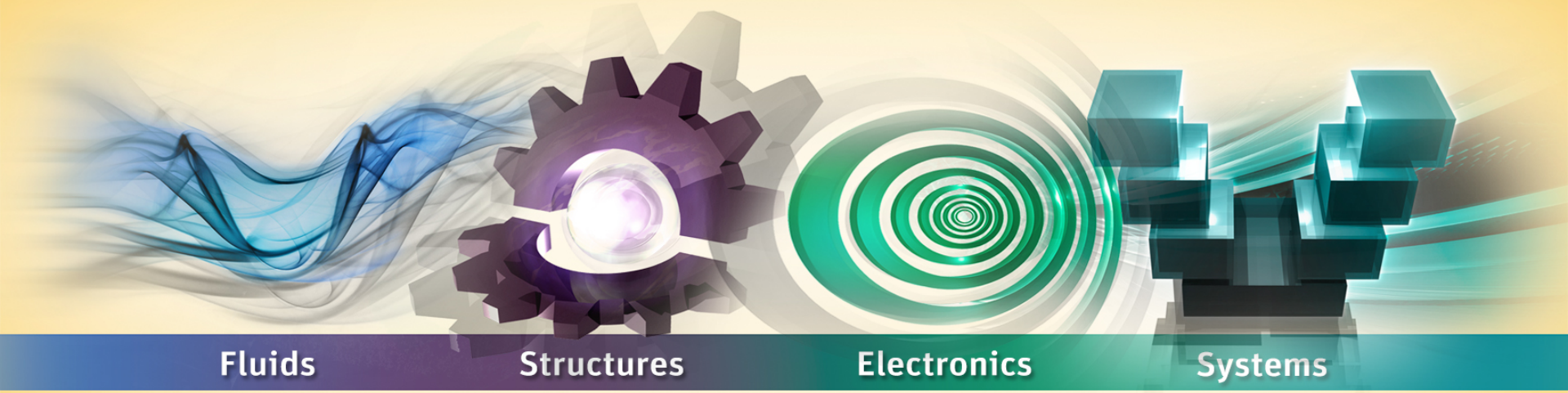
ANSYS

Realize Your Product Promise®

SCADE Training -
SCADE Suite Design Verifier



ESTEREL Technologies

Training Program

Introduction

Getting started with SCADE Suite Design Verifier

Writing properties

Verification with data

Design Verifier Methodology

Training Objectives

Learn how to integrate formal verification into your SCADE Suite development process

Perform formal verification on your SCADE design with Design Verifier

Training Exercises and getting used to the Design Verifier

INTRODUCTION

The Picture

Increasing importance of information technology in modern industrial systems:

- Transports: avionics, automotive, train interlocking control, etc.
- Communication: Secure networking, telephone systems, protocols, etc.
- Energy: nuclear plants

➔ **Safety and Mission Critical systems**

The Need for Correctness

System reliability increasingly depends on the correct functioning of hardware and software

Errors, or BUGS, can be fatal and very costly

- Safety-critical systems (Avionics, Nuclear plants, etc.)
- Products subject to mass-production (Processors, Consumer electronics)
- Large systems (Telephone switching, Baggage handling at airports)

➔ Increasing effort in reactive system validation

Classical Verification Techniques

System specification inspection and algorithms reviewing

- Static and manual techniques

➔ Ambiguous left parts of V cycle

Simulation & Testing

- Execute the design with some inputs and observe the outputs
- Input may be large, if not infinite.

➔ Coverage analysis required

Result and cost

- Reveal the presence of errors, not their absence
- 30 to 50% of project cost, may be more!

➔ Not exhaustive: bugs remain (corner cases)!

The Verification Bottleneck

Increasing cost of the verification phase

Time effort tends to be much greater than time system construction

Catching bugs:

THE SOONER, THE BETTER

Specification verification

Implementation verification

The Formal Verification Concept

Aim: Add mathematical reasoning into the verification processes

Goal: Reduce verification cost increasing system reliability

How: Add completeness to classical testing to detect uncovered bugs

Challenge: smooth and early integration in classical design methodologies

The Formal Verification Recipe

Provide a mathematical representation of Program Models – *A piece of Art!*

- Mathematical model that characterizes the system behaviors and properties

Provide a set of mathematical techniques to check program properties – *Another piece of Art!*

- Express the requirements as logical property formula
- Check the validity of the formula in the mathematical model

The Formal Verification Problem

Does my design model fulfill all of its requirements?

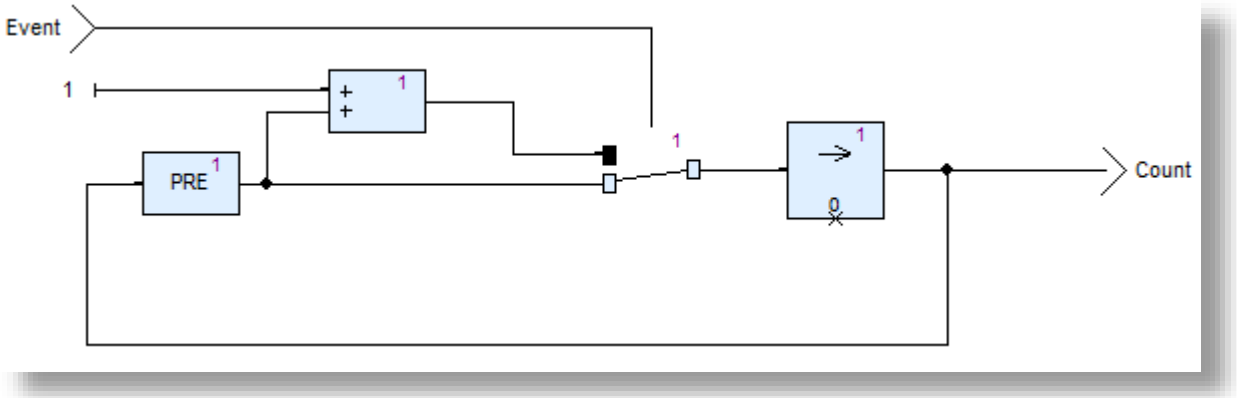
Design Verifier helps you verify the safety requirements:

- "The elevator will never move while the doors are open"

Checking a property is finding a proof that the property always holds for any execution of the design.

Formal Verification in SCADE

A SCADE model is mathematically defined as a set of data-flow equations with memory



$$\begin{cases} Count(0) = 0 \\ \forall t > 0,\ Count(t) = \begin{cases} Count(t-1) + 1, & if\ \ Event(t) = true \\ Count(t-1), & otherwise \end{cases} \end{cases}$$

We can prove mathematically: *for all t,*
Count(t) always greater or equal than Count(t-1)

Formal Verification in SCADE

Design Verifier is the module that automatically checks properties

- Solves Boolean problems over expression involving data and time (cycles)
- Uses SAT-based algorithms to solve the Boolean problems mixing different theories (Boolean satisfiability with induction, decision procedures for data equations solving, constraint solving, etc.)
- Yes/No answer to the question whether a given property always holds or not.

Complement to the verification by simulation activities

Design Verifier

Verify that the software requirements respect the safety requirements

Express the properties using SCADE Suite

No test to write

Check the properties for all conditions and all possible executions

Particularly efficient for handling logic parts of the design (control flow, decision logic, SSM)

Based on Prover SL™ engine from Prover Technology

Design Verifier Features Summary

Early detection of bugs without writing tests

Checking high-level and low-level safety property requirements

Simple framework for property specification

Counter-example test generation for detected bugs

Automatic load of counter-examples in the Simulator for fast and efficient debugging

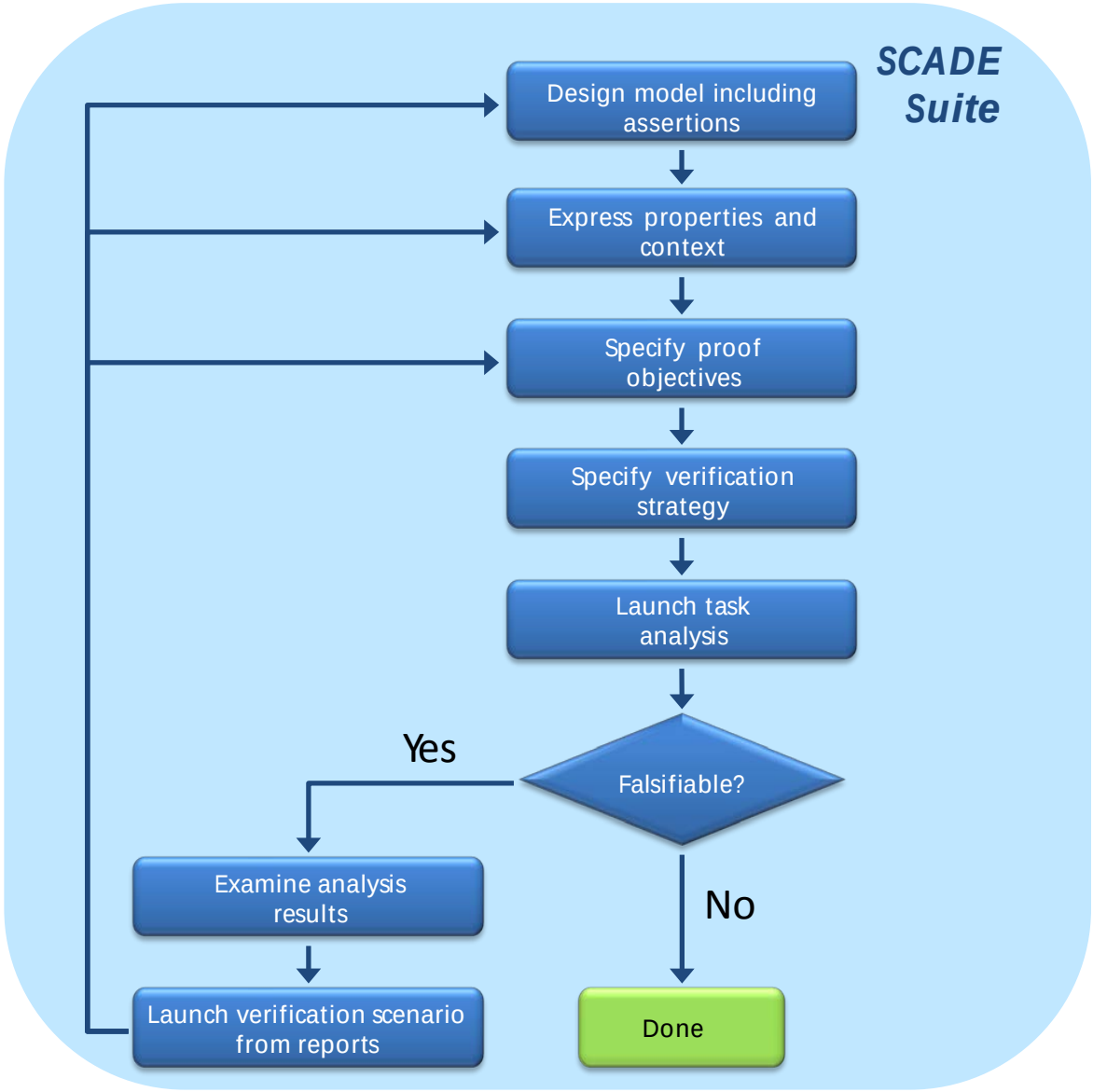
Successful Use of SCADE Design Verifier

Properties proved on several real industrial projects

- Aerospace & Defense: flight control application, sensor voter algorithms
- Automotive: embedded applications. Non-trivial bugs found in some cases with minimal length of tens of cycles
- Railways: started formal verification in current industrial projects

GETTING STARTED WITH DESIGN VERIFIER

SCADE Suite Design Verifier Workflow



Defining Properties

Identify the safety requirements in the software requirements

Focus on requirements addressing the logical and control flow aspects of the design

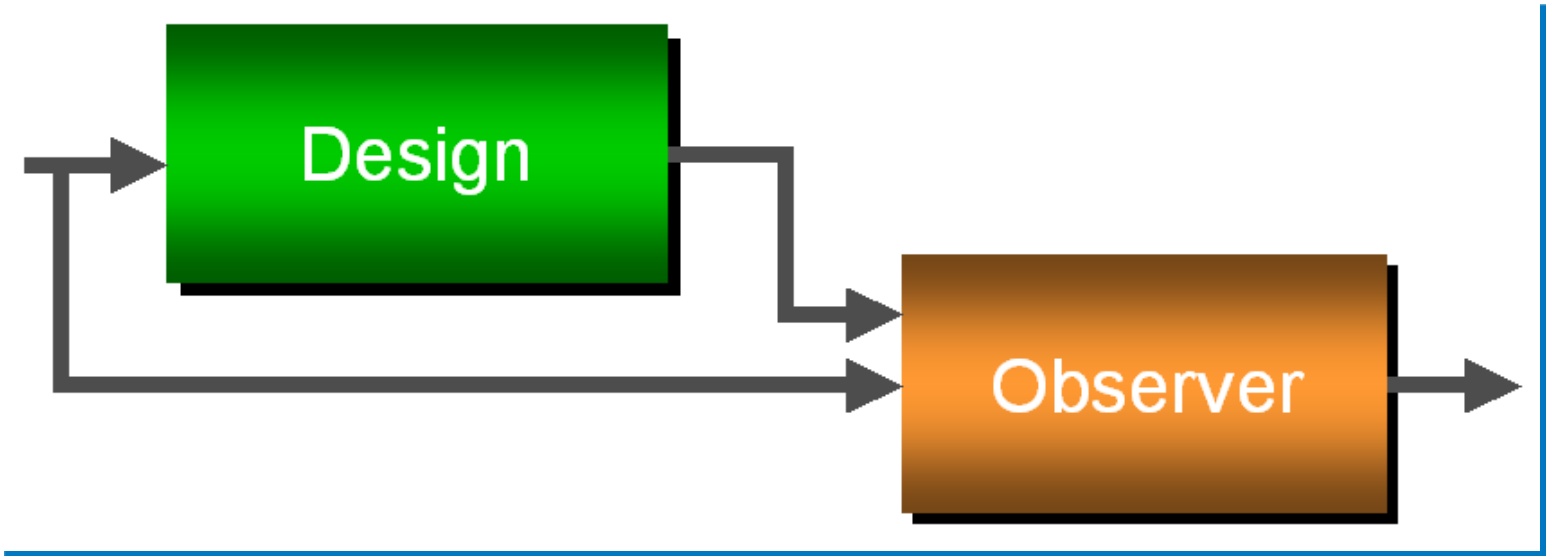
Express the properties as SCADE Suite operators called observers

- *"If A is true, then B must be true"*
- *"If X's value is 0, then after 3 cycles B must be equal to 1"*
- *"Never A and B at the same time"*

Properties as SCADE Observers

A property is implemented in a SCADE operator called an Observer.

- As inputs, it receives the values the property focuses on.
- It has one output, which is true if and only if, the property is true

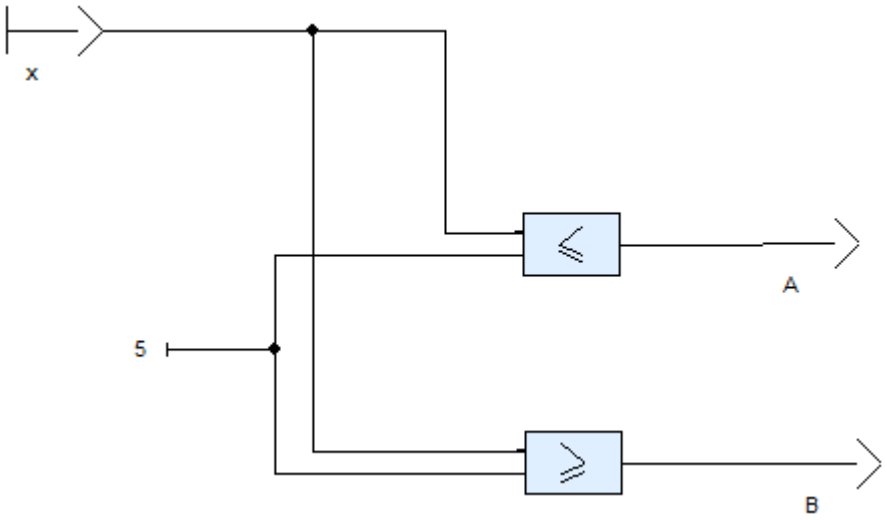


Exercise 01-1

Design model including assertions

1. **Go to folder "Prerequisite\Exercise 01-1" and open the project ValueCompare.vsw.**
2. **The Compare node: at each cycle**

- If x is lower than 5, then A is true, else it is false
- If x is greater than 5, then B is true, else it is false



Exercise

Exercise 01-1

Express properties and context

3. **Verify the specification requirement**
   "A and B should never be both true in the same cycle"

4. **Write a SCADE Suite Observer for the property**
   - *Input: A, B*
   - *Output: a Boolean flag that is true when (A and B) is false*

5. **Write a SCADE Suite operator CompareObserver that connects the operators Compare and Observer**

Exercise

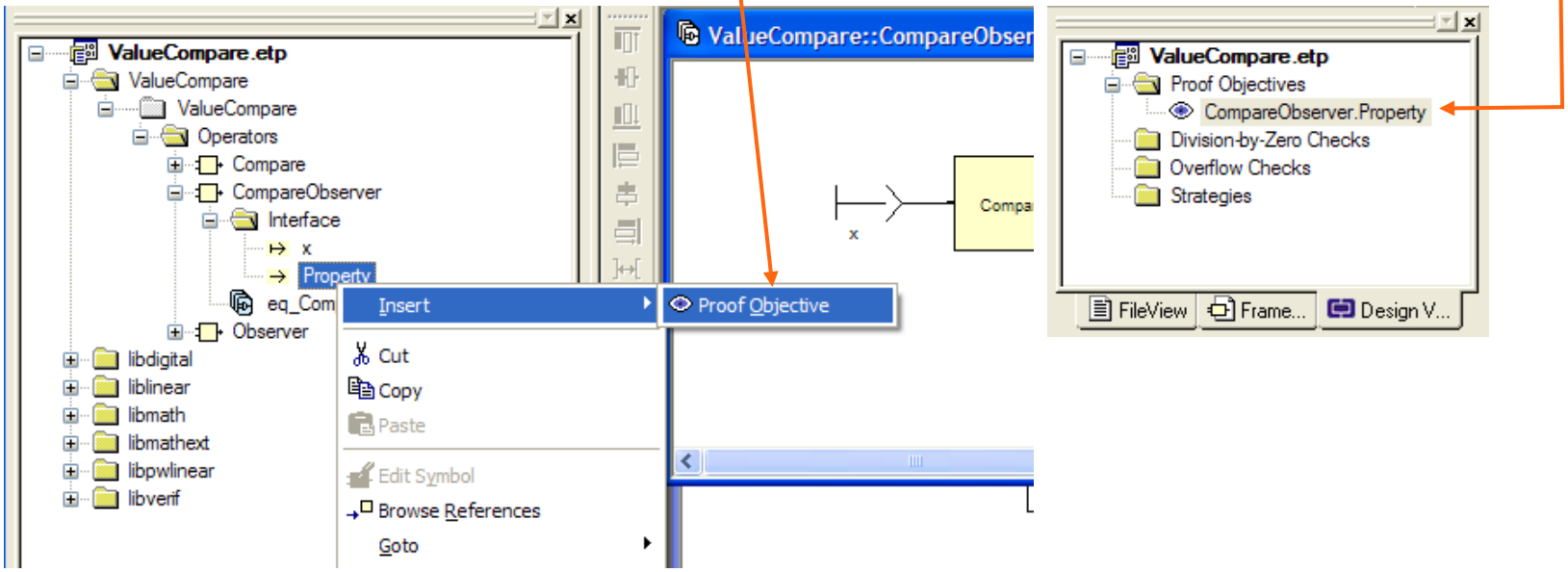
Exercise 01-2

Specify proof objectives and Launch task analysis

Exercise

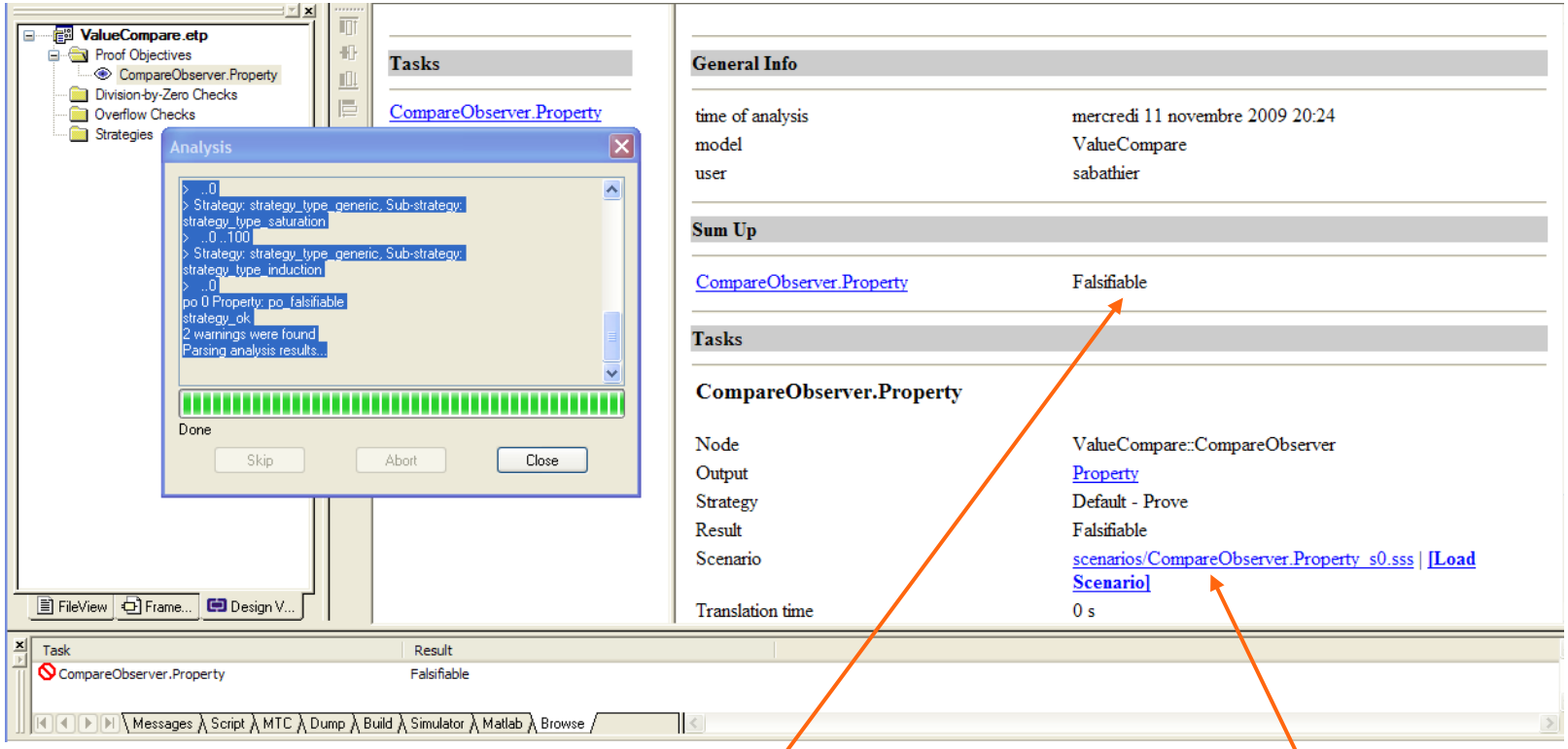
1. Declare the Boolean output A_and_B_Property as a new Proof Objective

2. Go to the Design Verifier tab, right-click on the Proof Objective and launch the Proof analysis



Exercise 01-2

Examine analysis results



The property is found falsifiable; a one-cycle length counter-example is generated with *x = 5*.

Read or play the generated counter-example in the Simulator

Solving the Problem

Correcting a design error

- We should clearly say which of A and B is exclusively true when x = 5

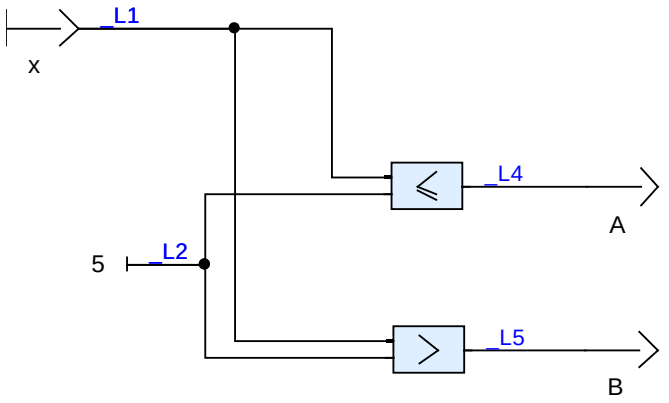
Adding environment information

- We know that the environment will never provide x = 5 as a realistic value
- (“Exercise 1\Step 3” sub-folder)

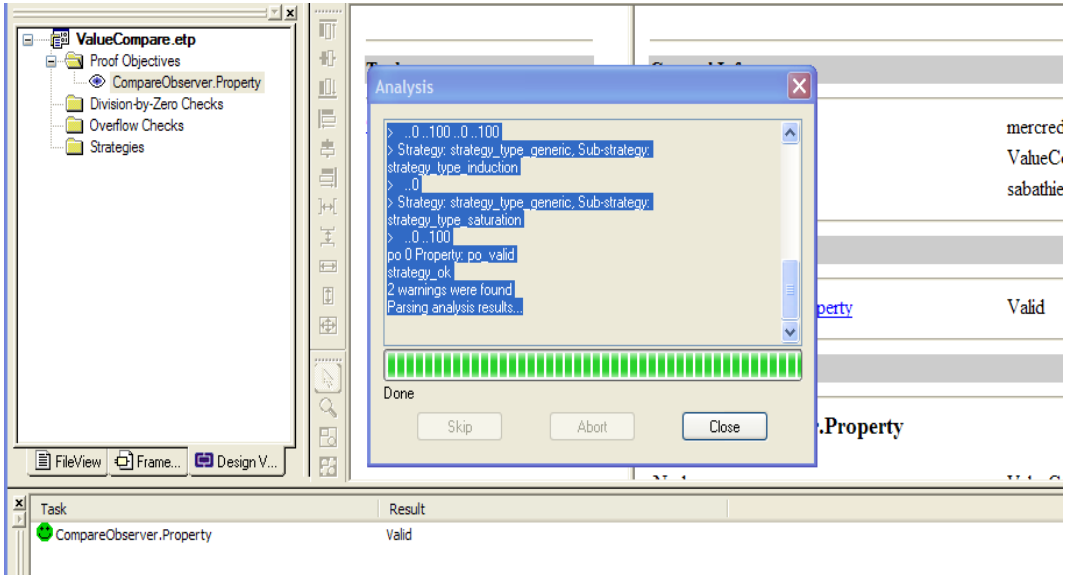
Exercise 01-3

Correcting the Design

1. **We decide that B is true whenever x is strictly greater than 5**



2. **Redo the verification shows that the property always holds**

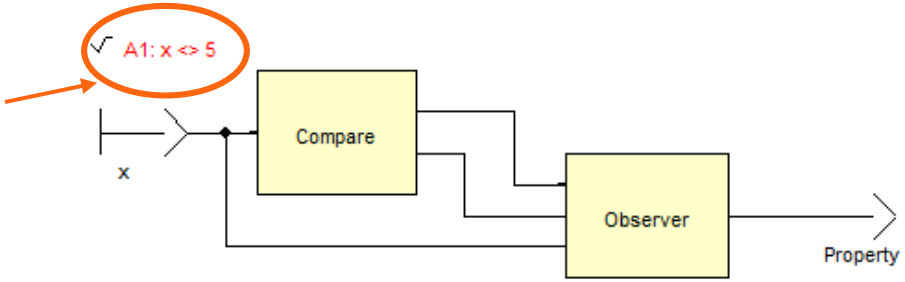


Exercise 01-4

Adding information

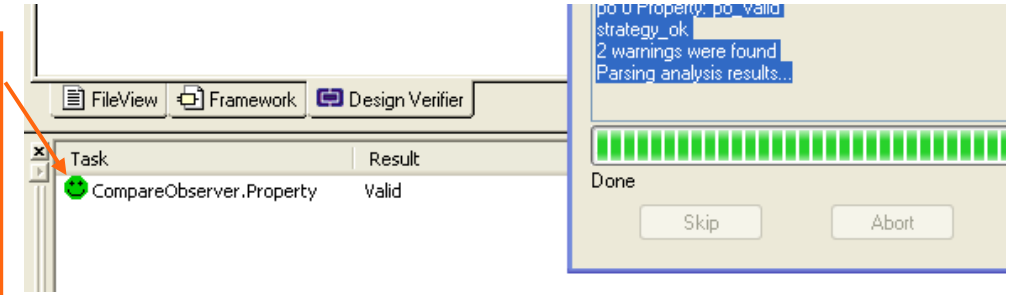
Exercise

1. **Add a SCADE Suite *assume* to mean that x is different from 5**



CAUTION: Design Verifier take into account only the SCADE assertions on root

2. **Redo the verification, the property is found valid**



Exercise 01-5

Write another Property

1. **From the original design, write a SCADE Suite Observer for the requirement:**

- "If A and B are both true, then x = 5"

2. **Redo the verification**

Exercise

WRITING PROPERTIES

SCADE Suite Observers

Use the SCADE Suite language to write nodes whose output is Boolean, that should always be true:

- Boolean logic:
  - Prop = (A and B) or not(C)
- Relational expressions
  - Prop =((X = 5) and (Y > 0)) or (Z > X/3)
- Temporal expressions
  - Prop = true -> not(pre(B and (X = 0))) or (Y = 1)

The SCADE Verification Library

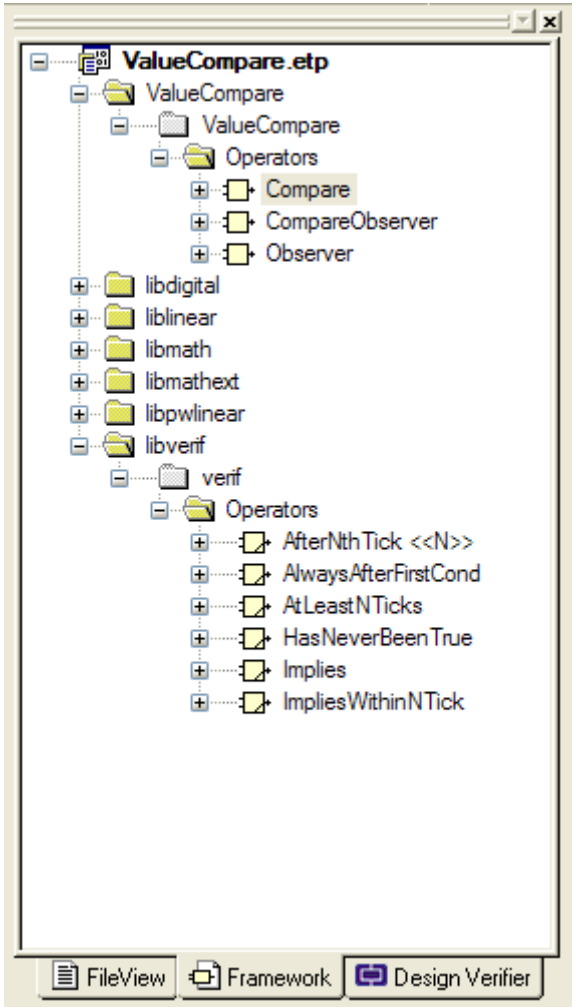
To help the writing of properties

4 patterns

- AlwaysAfterFirstCond
- AtLeastNTick
- HasNeverBeenTrue
- Implies

2 families of patterns

- AfterNthTick
- ImpliesWithinNTick



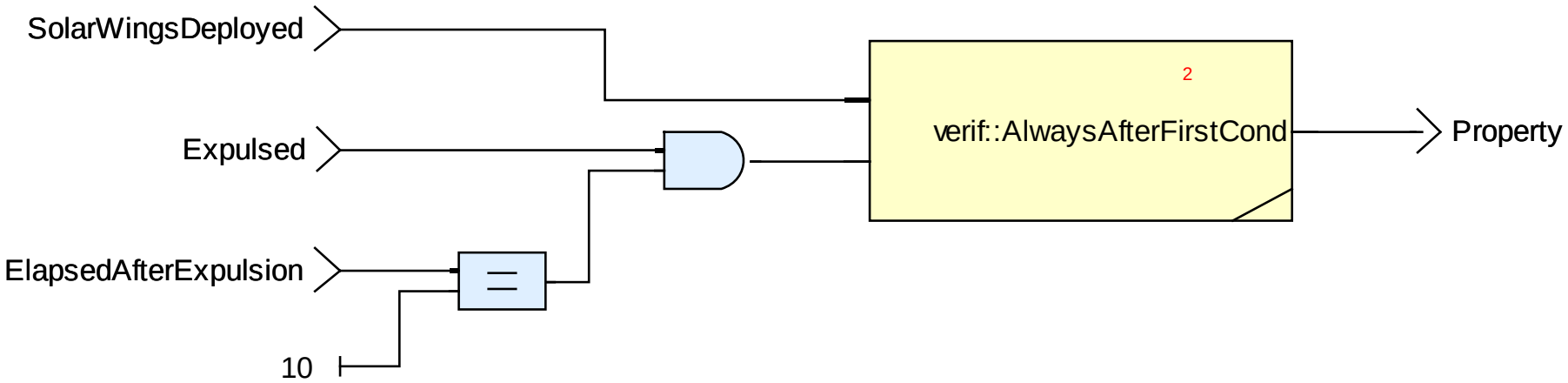
Pattern: AlwaysAfterFirstCond

Equal to its input Input1 once Cond becomes true. True before that time.

Cycle	1	2	3	4	5	…
Cond	false	false	true	*	*	*
Input1	false	false	false	true	true	false
Output1	true	true	false	true	true	false

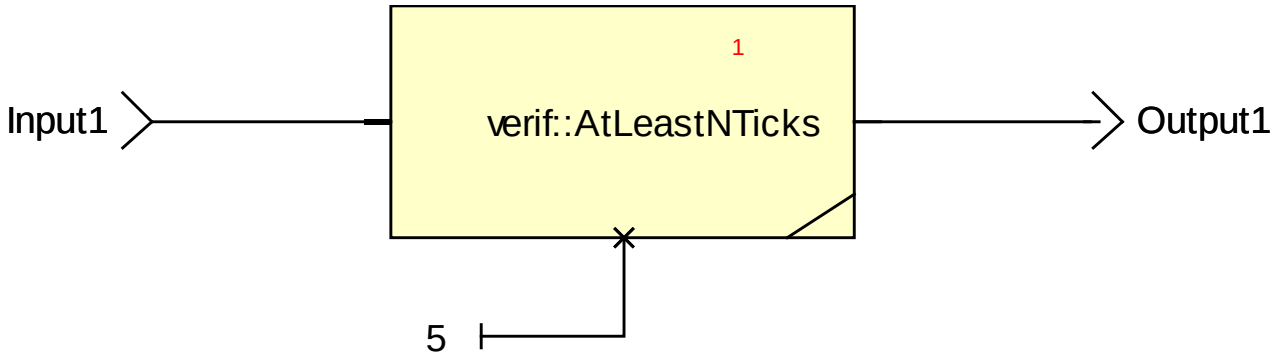
AlwaysAfterFirstCond: Example

The satellite wings should always remain deployed after 10 cycles after expulsion



Pattern: AtLeastNTicks

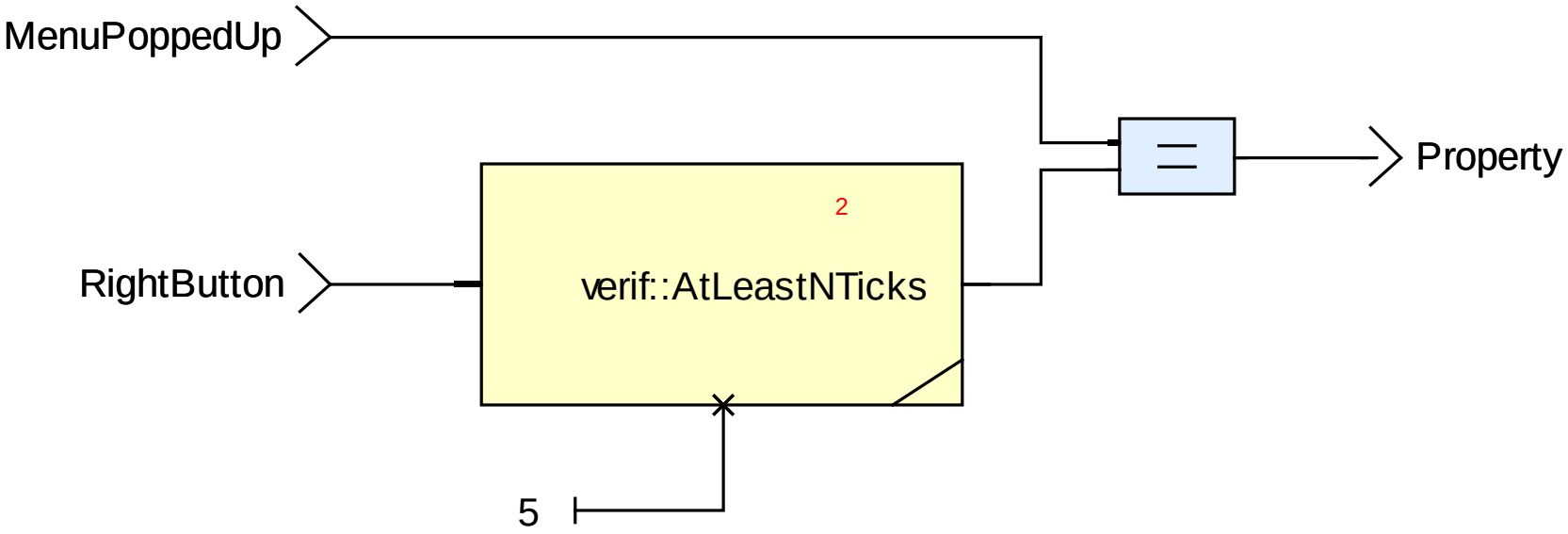
Equal to its input condition each time this condition has been true N times.
False before that and each time its input condition is false.



Cycle	1	…	4	5	6	7		10	11
Input1	true		true	true	false	true	…	true	true
Output1	false	false	false	true	false	false	false	false	true

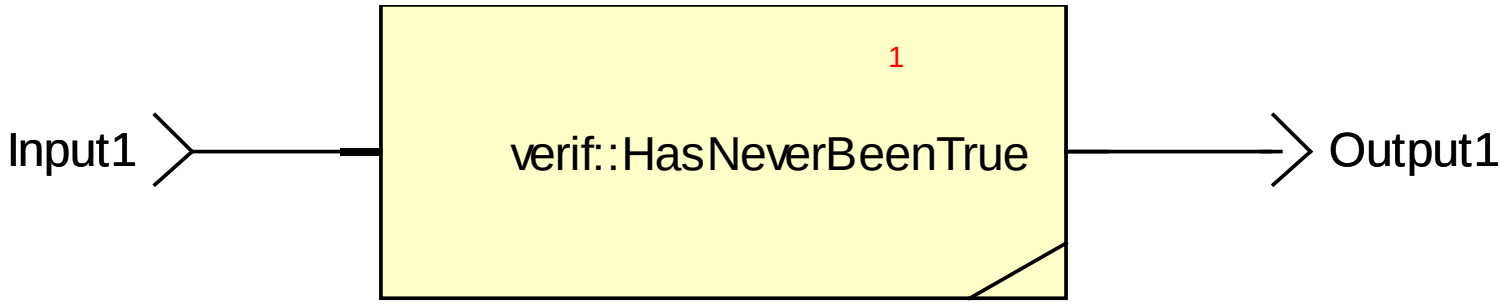
AtLeastNTicks: Example

If *RightButton* is pressed during 5 cycles then Menu is popped up until *RightButton* is not pressed anymore



Pattern: HasNeverBeenTrue

Becomes false as soon as its input turns to true first time. True before that time.



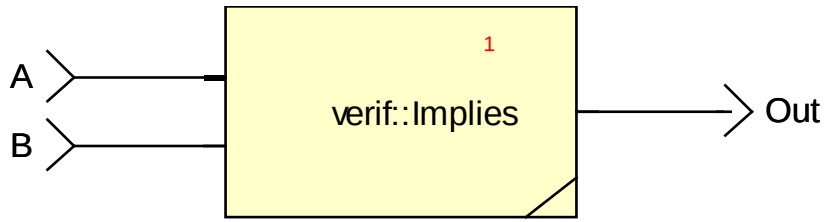
Cycle	1	2	3	4	5	…
Input1	false	false	true	true	*	*
Output1	true	true	false	false	false	false

Pattern: Implies

Commonly used predefined node logical patterns: e.g. “Implies” (➔)

“If A is true then B is true”:

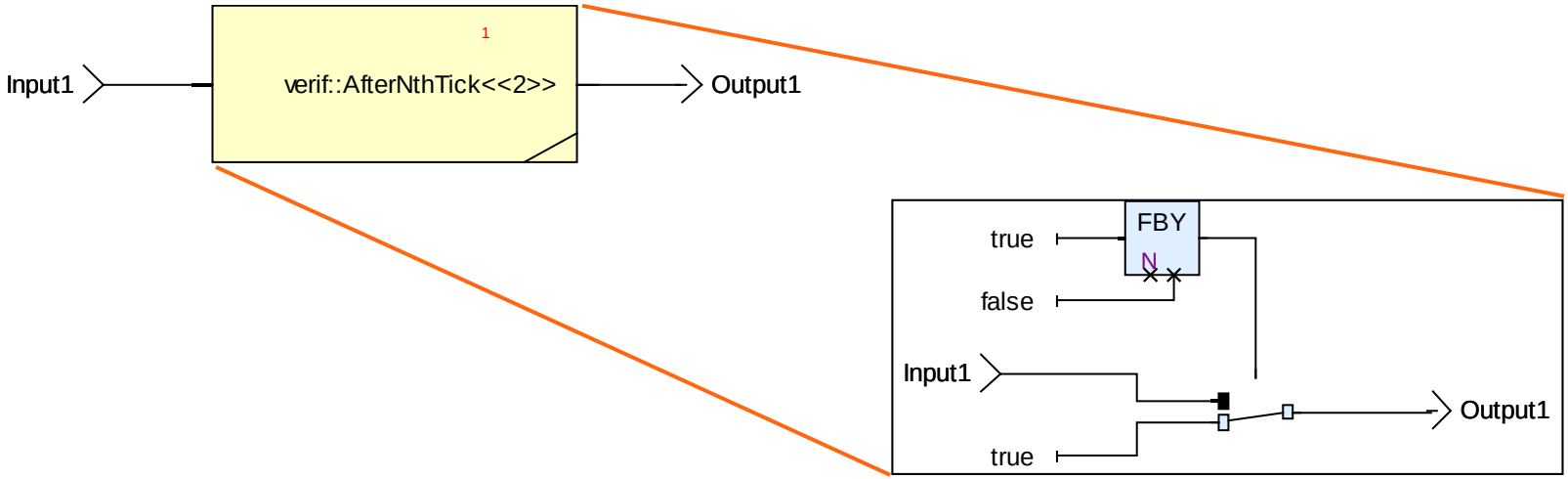
not(A)or B = A ➔ B



A	B	Out
true	true	true
true	false	false
false	true	true
false	false	true

Family Pattern: AfterNthTick

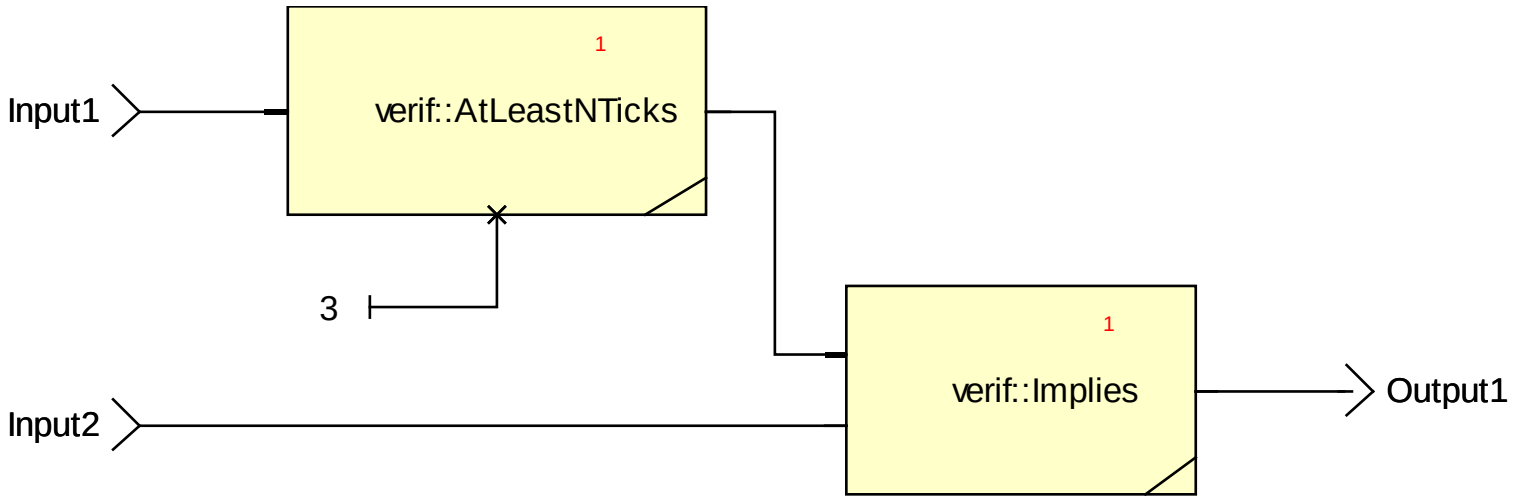
Always equals to its input condition after N cycles. True before that.



Cycle	1	2	3	4	5	…
Input1	true	false	false	true	true	false
Output1 (N=2)	true	true	false	true	true	false

Family Pattern: ImpliesWithinNTick

Output1 is “Input1 implies Input2” when Input1 has been true N times. True before that.



Exercise 01-6

Using a Property Pattern

1. Re-write the property of Exercise 01-5 using the Implies pattern

2. Redo the verification

Exercise

VERIFICATION WITH DATA

Arithmetic Expressions

Supported data-types:

- Booleans
- Integers
- Fixed-size integers (expressed in bit size)
- Rational numbers

Operations

- All Boolean operators (not, and, or, etc.)
- All arithmetic operators (+, -, *, /, %, etc.)
- All relational operators (=, <, >, <=, >=, etc.)
- Linear arithmetic only

Supported Arithmetic

Integer arithmetic: multiplication with at most one variable

Rational arithmetic: multiplication and division with at most one variable

Supported	Not supported
`5*X + Y`	`X*Y + 1`
`Z/2.0 – W`	`Z div (if X=0 then 1 else X) + 5`
`(T%3) * 2`	`(W%P) * 3`

Trigonometric and power (exponential as well) functions are not rational functions. This is the reason why they are not supported.

Exercise 02

Solving Equations

Exercise

1. **Linear equation system**

$$\begin{cases} a.x + b.y = e_0 \\ c.x + d.y = e_1 \end{cases}$$

2. **Use Design Verifier to solve the system**
   - Try with a=2, b=4, c=5, d=10, e0=6, e1=15
   - Find X and Y using Design Verifier
   - Steps:
     - *Model the equation system*
     - *Model a property to find out a solution*

Real Numbers

Classical gap between mathematical real numbers and their floating-point implementation

Since floating-point implementation approximates the real numbers

- A proven property by Design Verifier can be falsified in the Simulator
- A falsifiable property by Design Verifier may not be so in the Simulator

Problem occurs when some comparison involve a real number

- Boolean values computed out of real numbers arithmetic can differ between the model and the implementation

Exercise 03

Exercise

1. **Input: X,Y constant real numbers with X < Y**

2. **Load the Prerequisite\Exercise03\Zeno.xscade SCADE design that computes:**
   - R(0) = X
   - R(t) = R(t-1) + ((Y – R(t-1)) / 2)

Example
X = 1.0 and Y = 2.0
Simulate for 55 cycles

3. **Write and analyze the property**
   - For all cycle t, (X < Y) implies (R(t) < Y)

4. **Simulate the design for a certain number of cycles**
   - What do you find out?
   - Conclude

Exercise 04

Verifying a Sorting Algorithm

1. **Open the project in Prerequisite\Exercise 4\Sort.etp**

2. **The Sort operator sorts by increasing order an array of N values given as input and produces the result in an output array**

3. **Verify with Design Verifier that the Sort operator is correct**
   - Define the property for N=5
   - Increase N
   - Conclude

Exercise

Exercise 05

Verifying a FIFO

1. **Open the project in Prerequisite\Exercise 5\Fifo.etp**

2. **The FifoController operator models a FIFO**

3. **Verify with Design Verifier that the following properties of a FIFO**
   - Overflow and Underflow detection
   - Bypass is performed correctly

Exercise

COMPLEXE ARITHMETIC AND IMPORTED FUNCTIONS

DESIGN VERIFIER METHODOLOGY

Design Verifier Strategies

Two generic strategies:

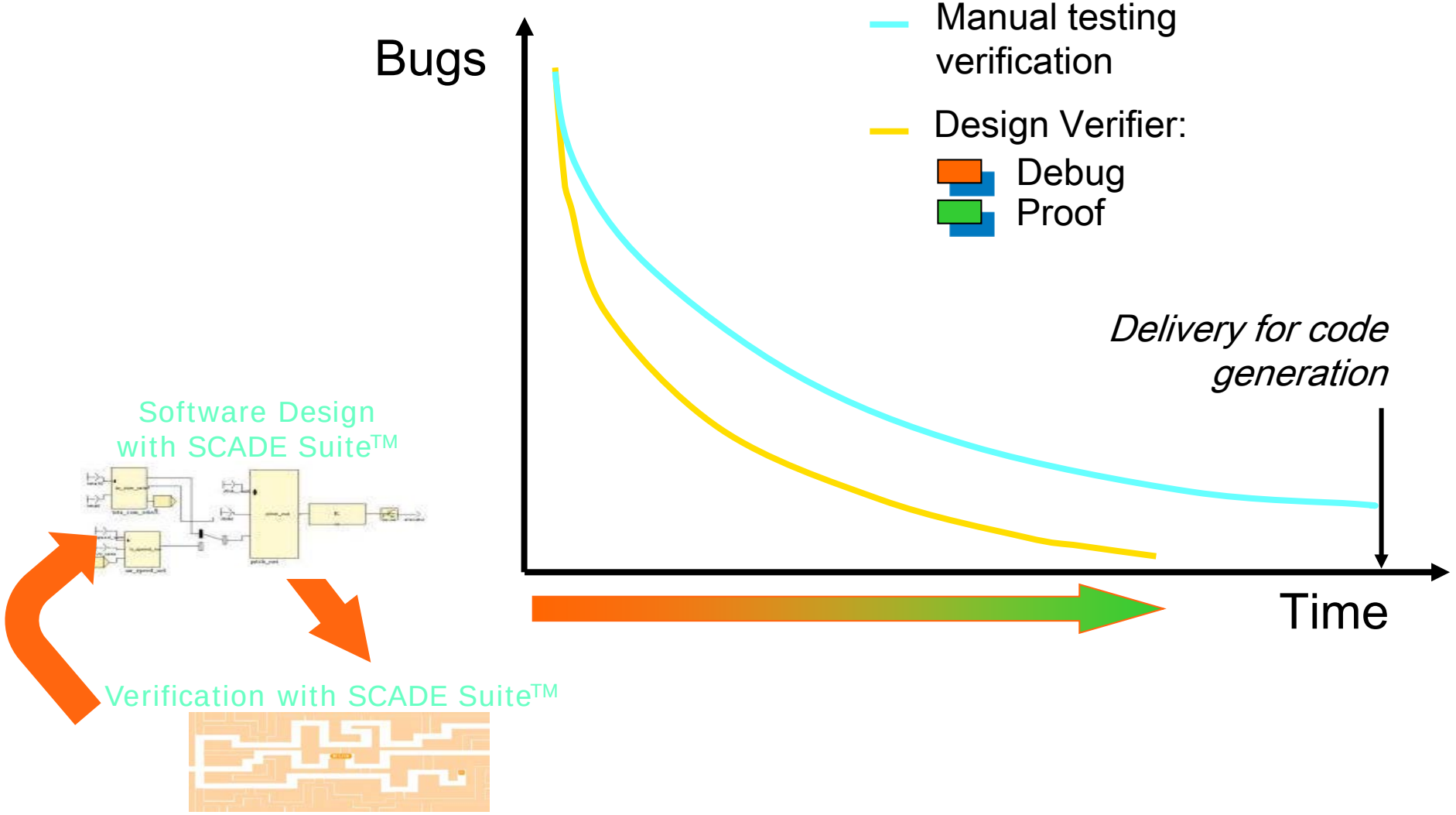
- Debug: Ask the Design Verifier whether the property is falsifiable within some fixed number of execution cycles. If yes, a counter-example is found quickly and generated for simulation.

- Proof: Ask the Design Verifier whether the property is always valid. If so, the property may be found valid. If not, a counter-example may be found. The result is indeterminate if the property cannot be proven.

➔ How to use the Design Verifier strategies?

➔ What to do when the result is indeterminate?

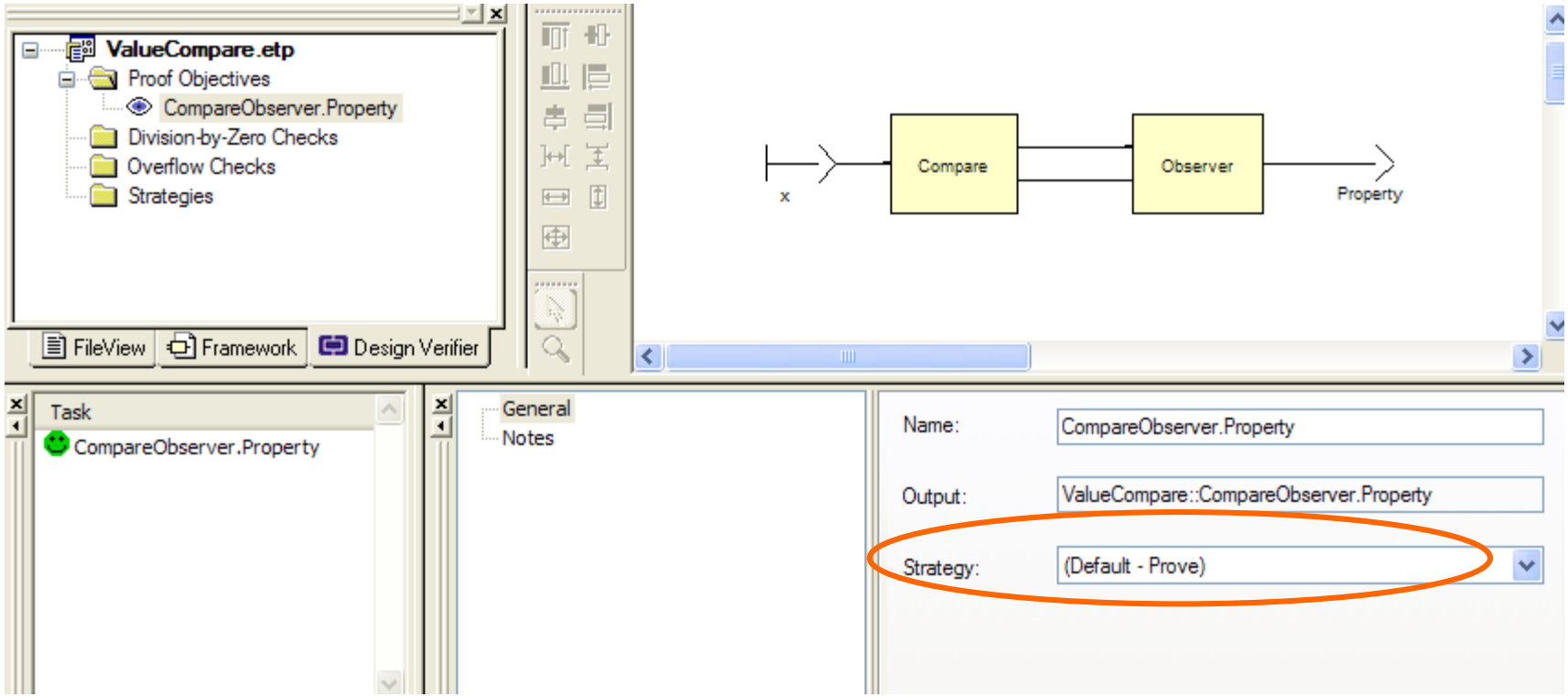
Strategy Usage



Setting Strategies (1/2)

Default strategy for proof objectives is the Prove strategy

- Select a proof objective and display its properties.

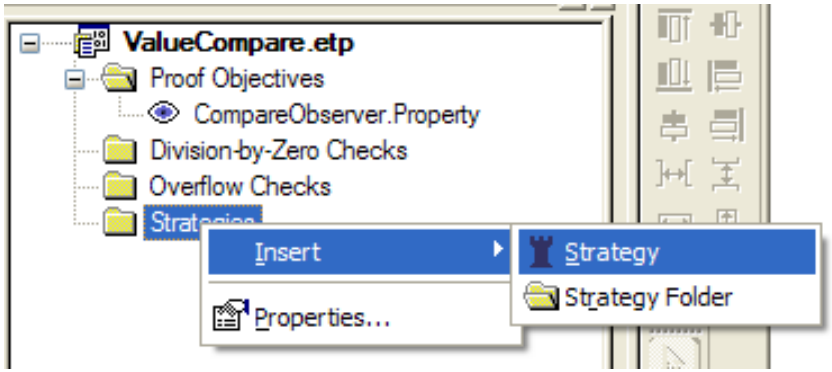


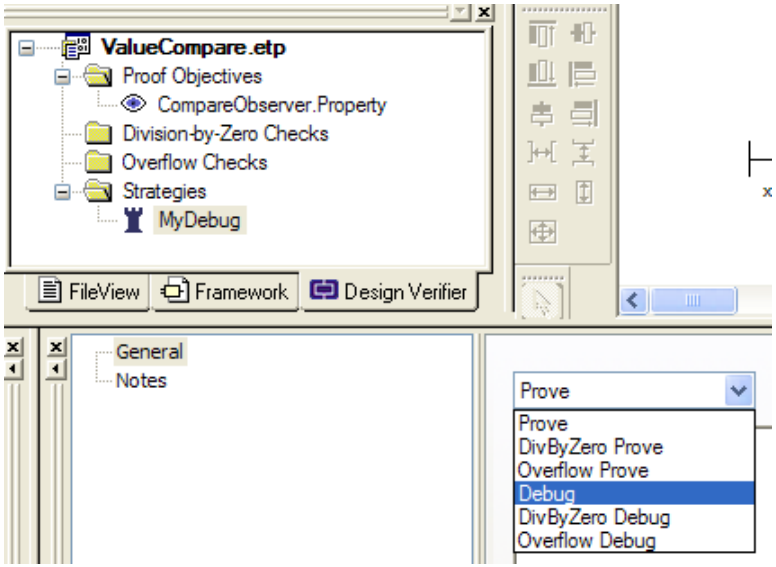
Setting Strategies (2/2)

To define a new strategy, e.g. a Debug strategy, right-click on the Strategies folder

Give the strategy a name, e.g. My Debug

Right-click on the new strategy to display its properties and choose the Debug strategy type

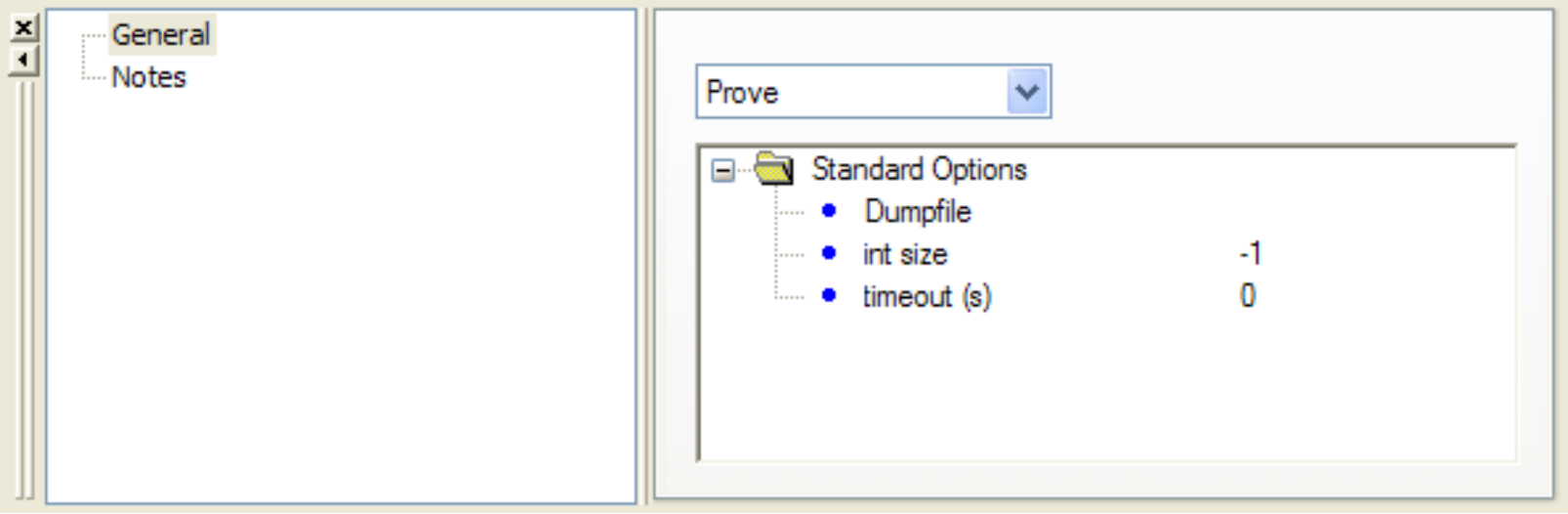




Strategies' Common Options

Timeout: amount of time in second allowed for analyzing the property (0: no timeout)

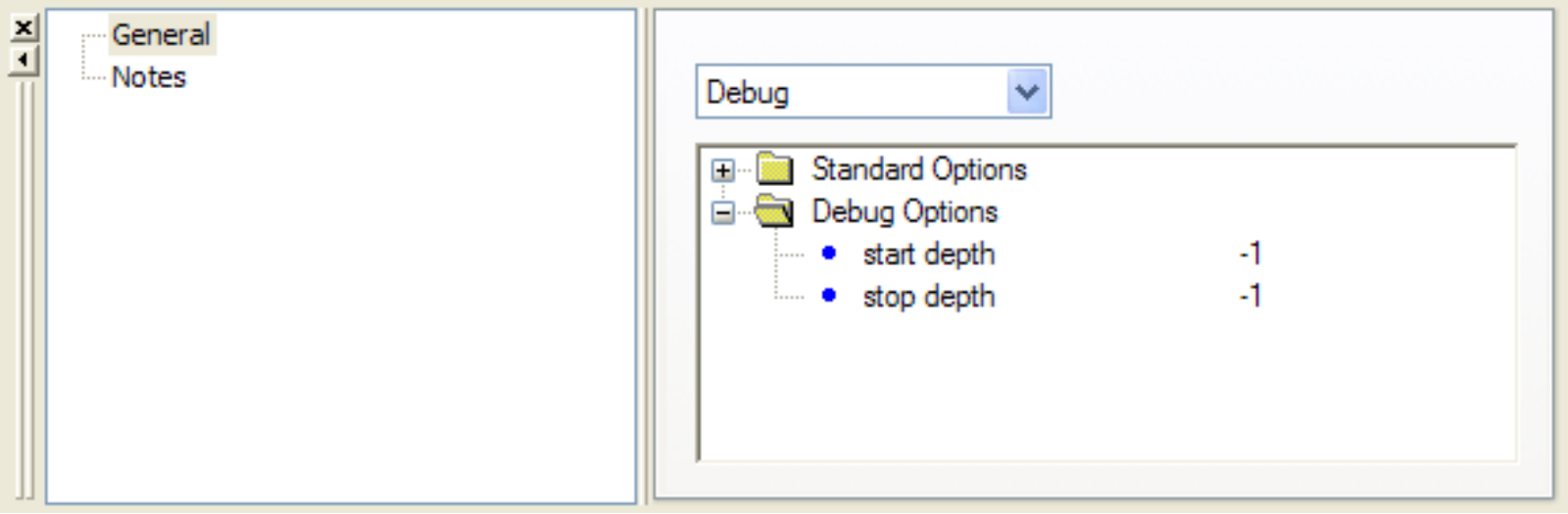
Integer size: bound on integer value expressed in number of bits for integer size (-1: no bound).



Debug-Strategy's Options

Start depth: cycle number at which the analysis starts. Default 0.

Stop depth: cycle number at which the analysis stops. Default 100



Strategy Usage

Use the Debug strategy in the early phase of design

Use the Proof strategy in the final phase

Benefits:

- Find bugs quickly, detecting soon specification ambiguities and problems
- Prove the correctness of the design with respect to the system requirements
- High confidence in the design
- Reduced verification cost

➔ But what do I do when I get indeterminate results ?

Debug Strategy Indeterminacy

Search for a bug up to a bounded number of execution cycles

Indeterminacy means that no bug exists up to the bound

Three cases:

- The bug is deeper: increase the bound
- There is no bug: try the proof strategy

Coping with Indeterminacy

Indeterminacy means in fact that Design Verifier could not prove the property because too hard to find.

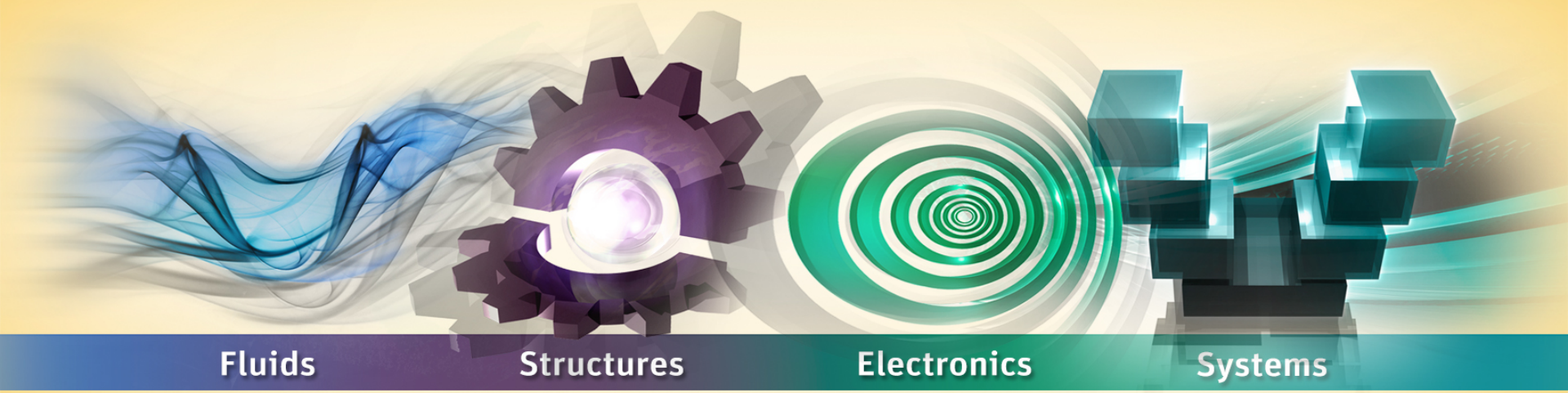
Note: When using the Induction strategy, Indeterminacy means that the problem is not "inductive", even if the model is simple and linear.

Problem	Way to solution
Large data-path domains	Add environment constraints (SCADE Suite assertions)
Non-linear expressions in data-path	Abstract non-linear constructs into linear forms (e.g sin as [-1,1], look-up table, etc.)

ANSYS

Realize Your Product Promise®

LABS



ESTEREL Technologies

Case Study: Step 1

1. **We want to verify that:**
   - When the Brake pedal is pressed (> PedalMin), the Cruise control must be in Interrupt or Off state

2. **Create a new project CruiseControlProof for Design Verifier inserting *Prerequisite\Lab\CruiseControl\CruiseControl\CruiseControl.etp* project**

3. **Create in a package ProofPkg, an operator Observer to verify the property.**

4. **Create an operator BrakeProperty where you plug the CruiseControl node with Observer node. Create the Proof Objective and analyze it**

Tip: Use the Implies operator of the libverif library provided under the SCADE installation directory

Lab

Case Study: Step 2

1. Explain the result if falsifiable

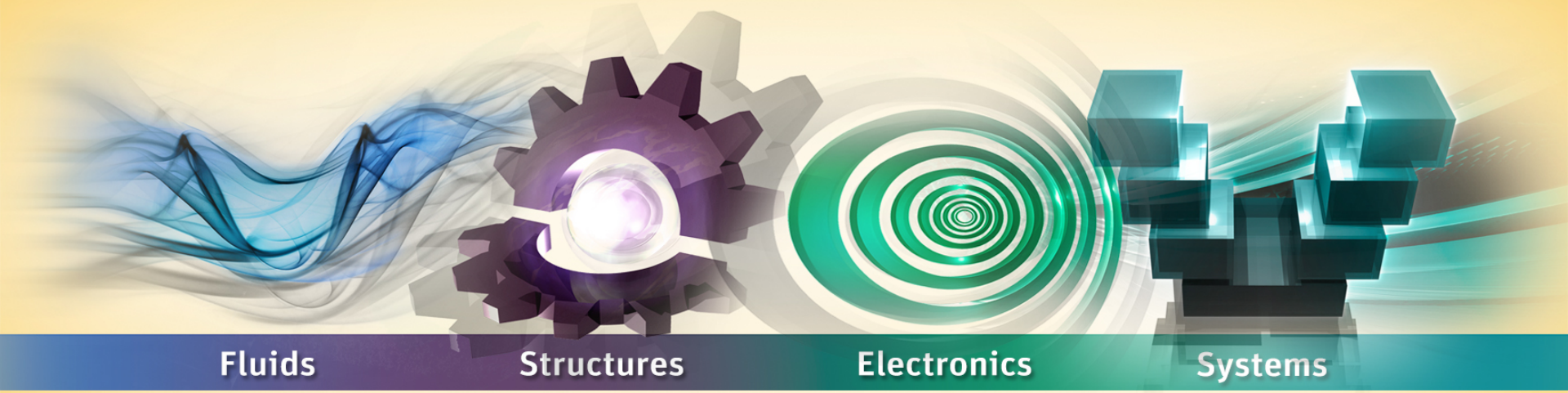
2. Find a solution to fix the problem

Lab

ANSYS®

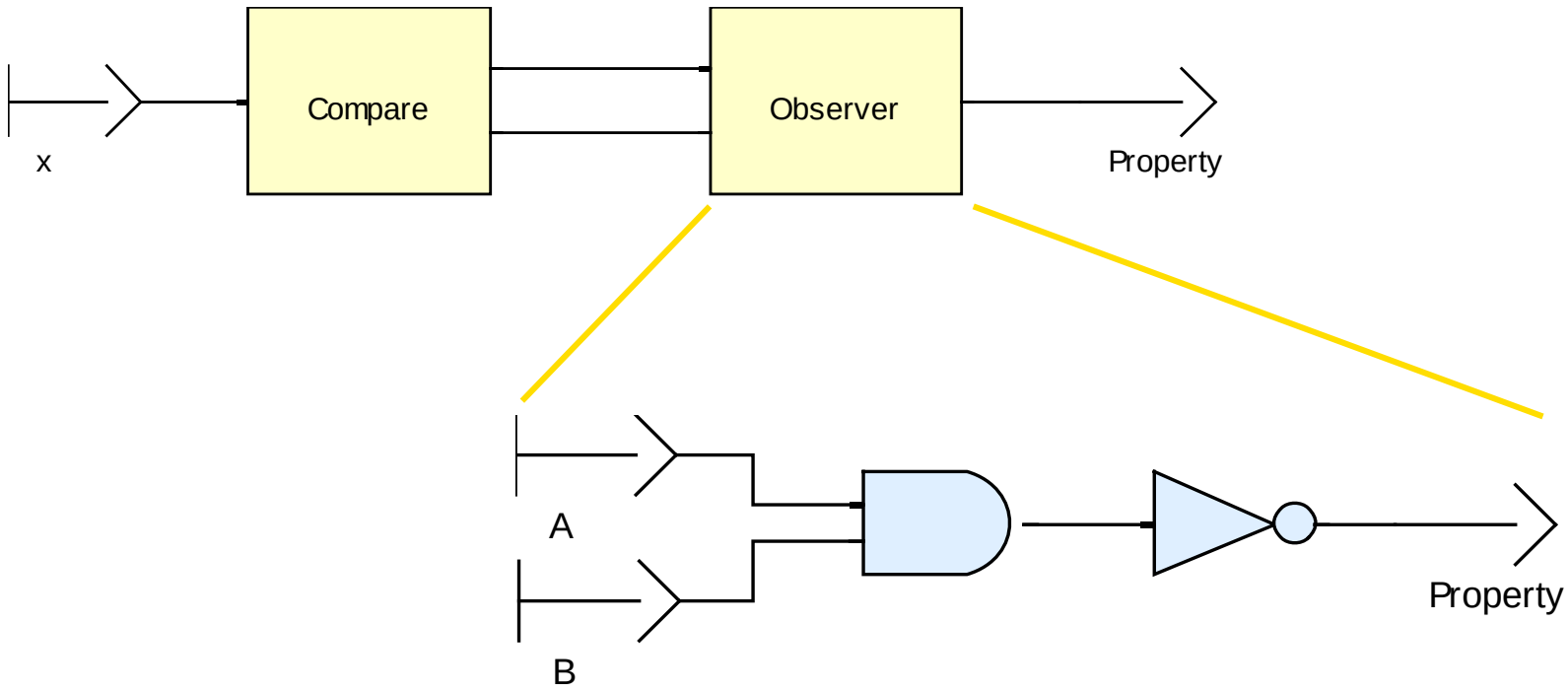
Realize Your Product Promise®

Exercises Solutions



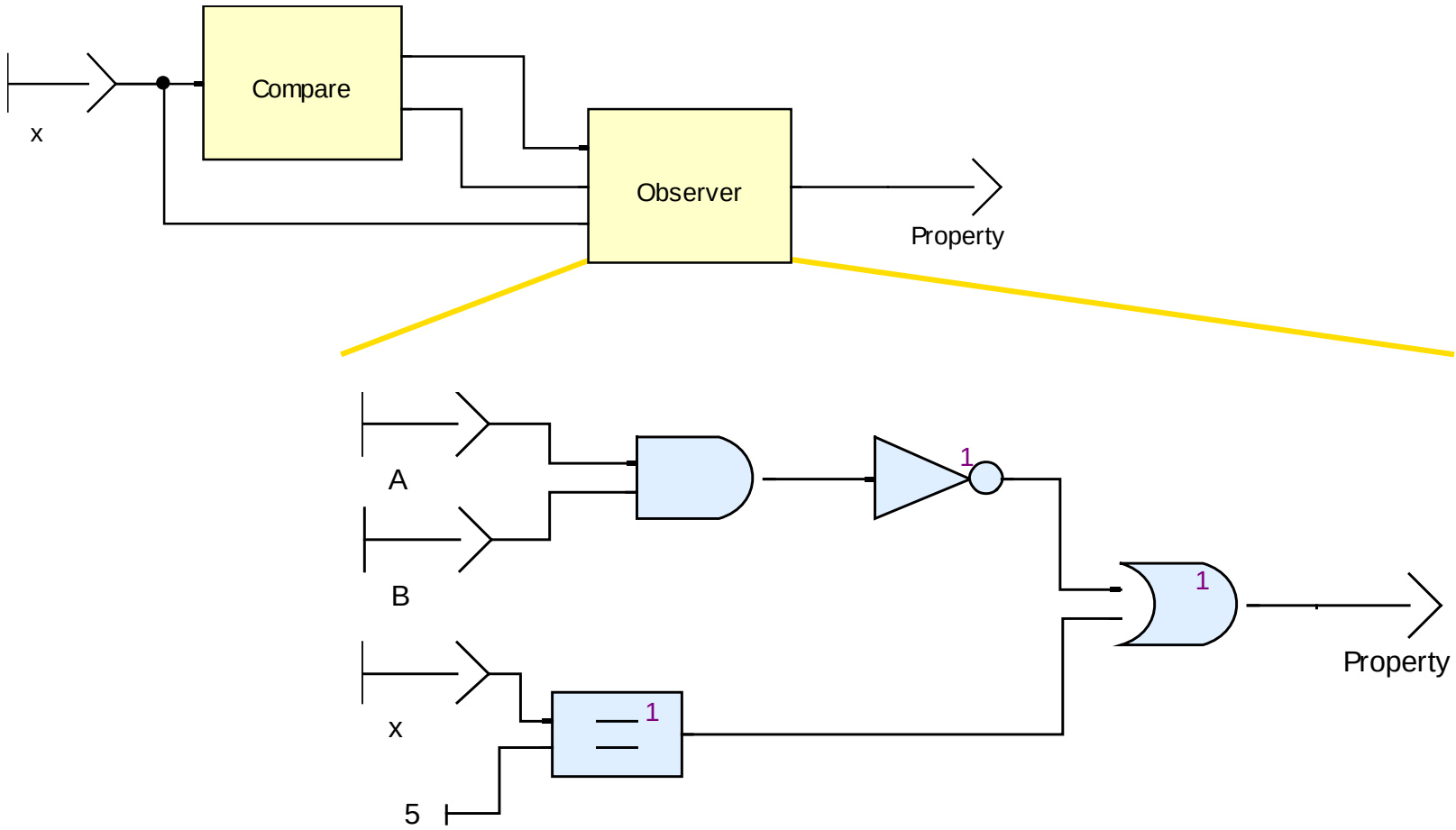
ESTEREL Technologies

Exercise 01-1



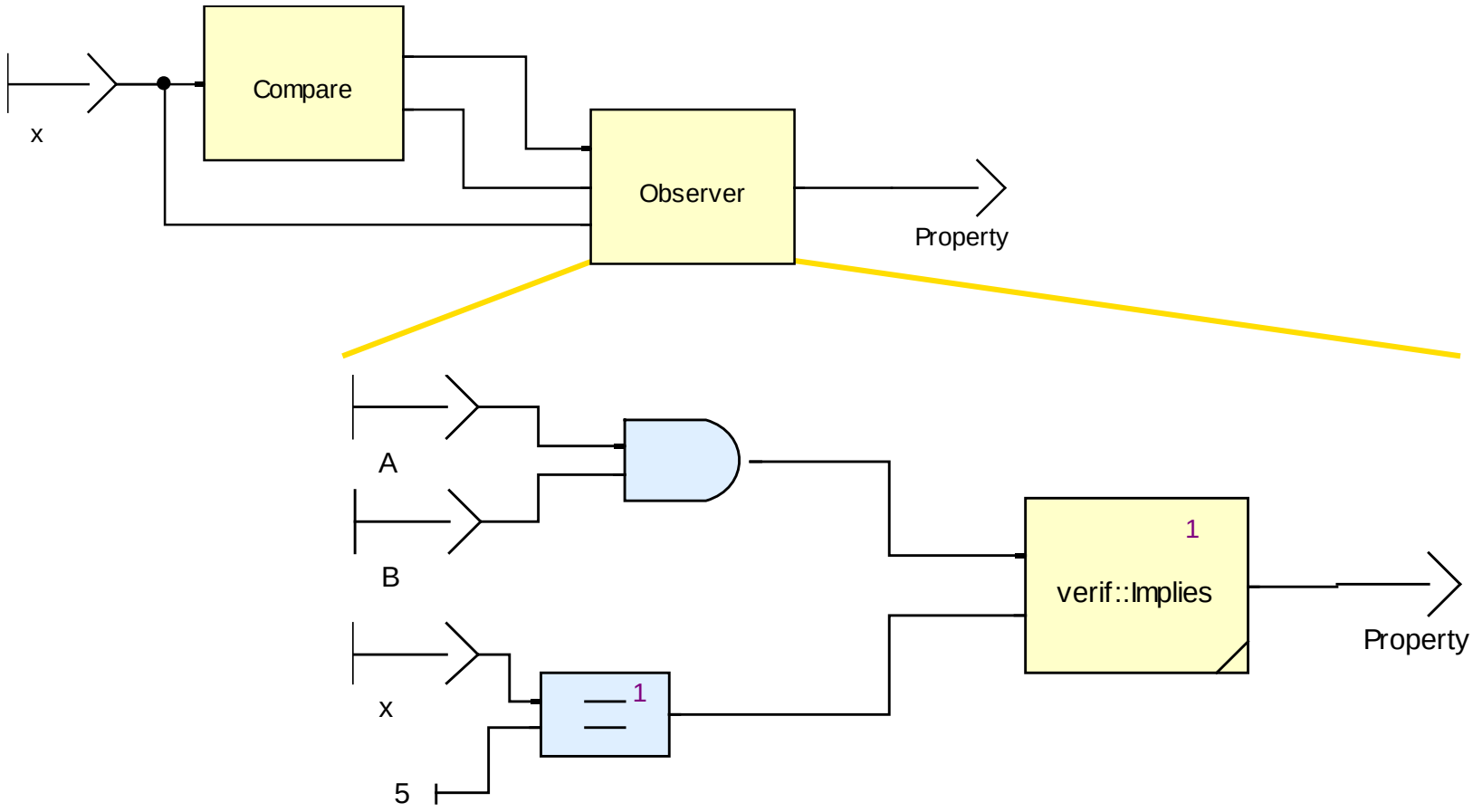
Exercise

Exercise 01-5



Exercise

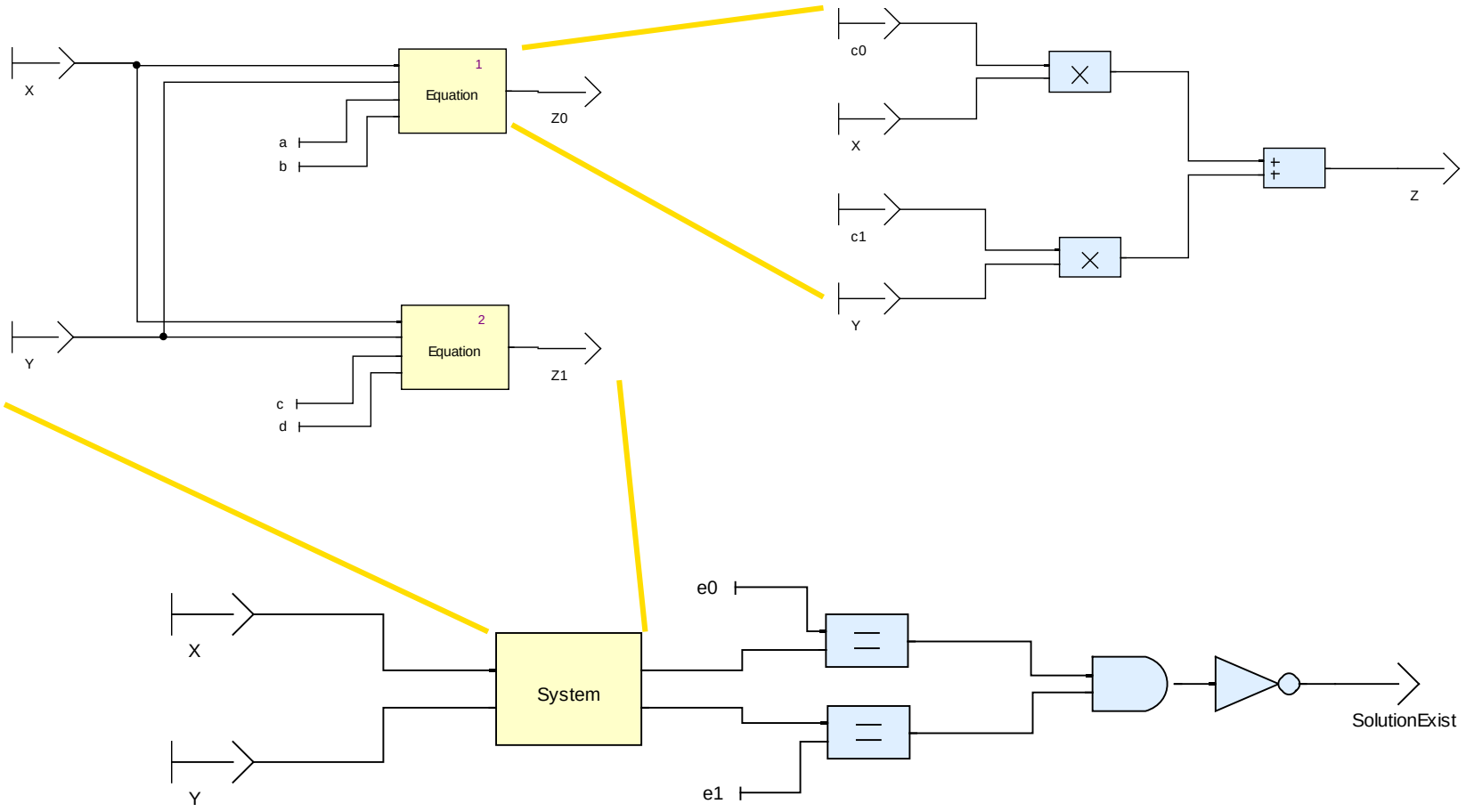
Exercise 01-6



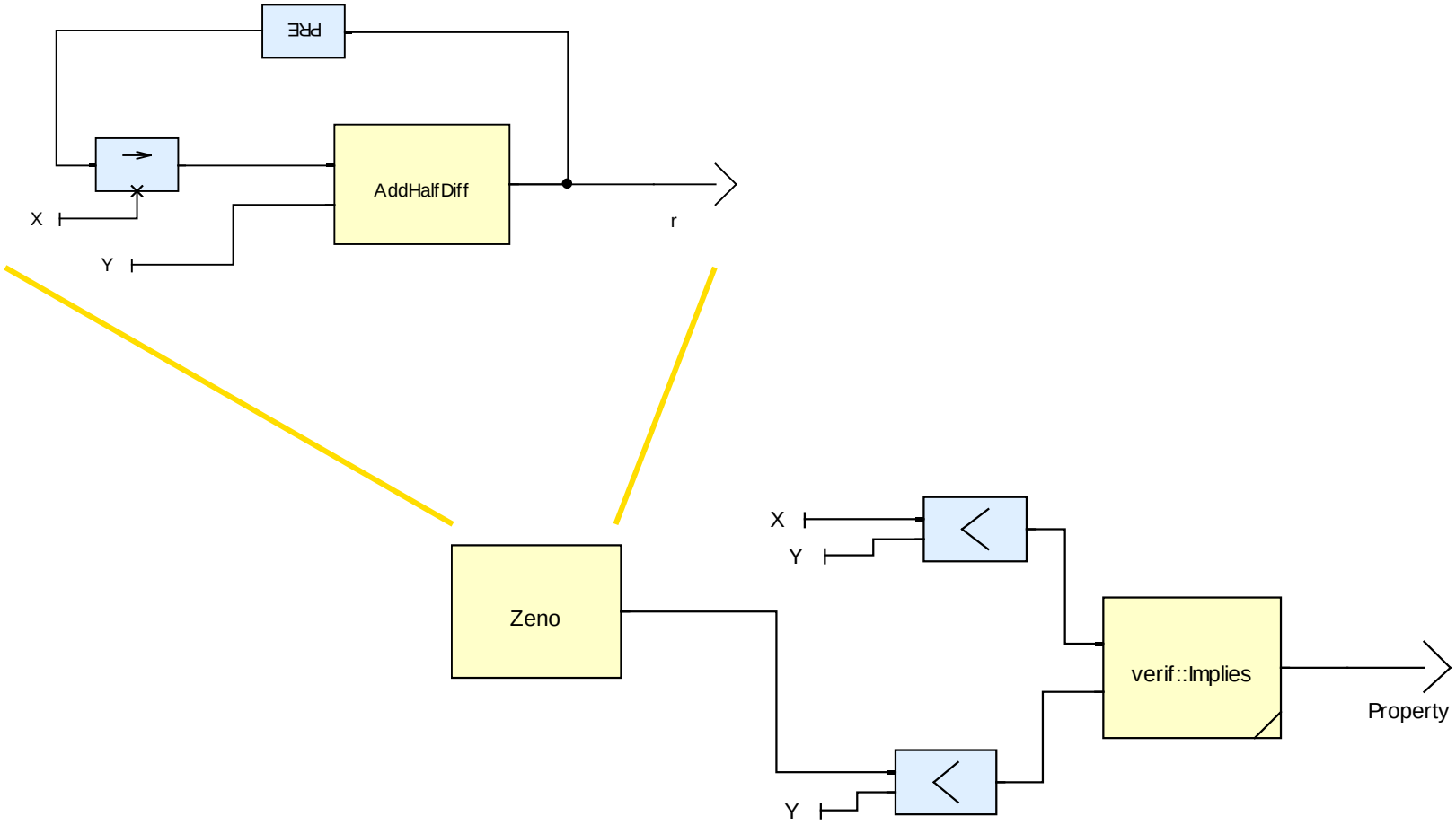
Exercise

Exercise 2

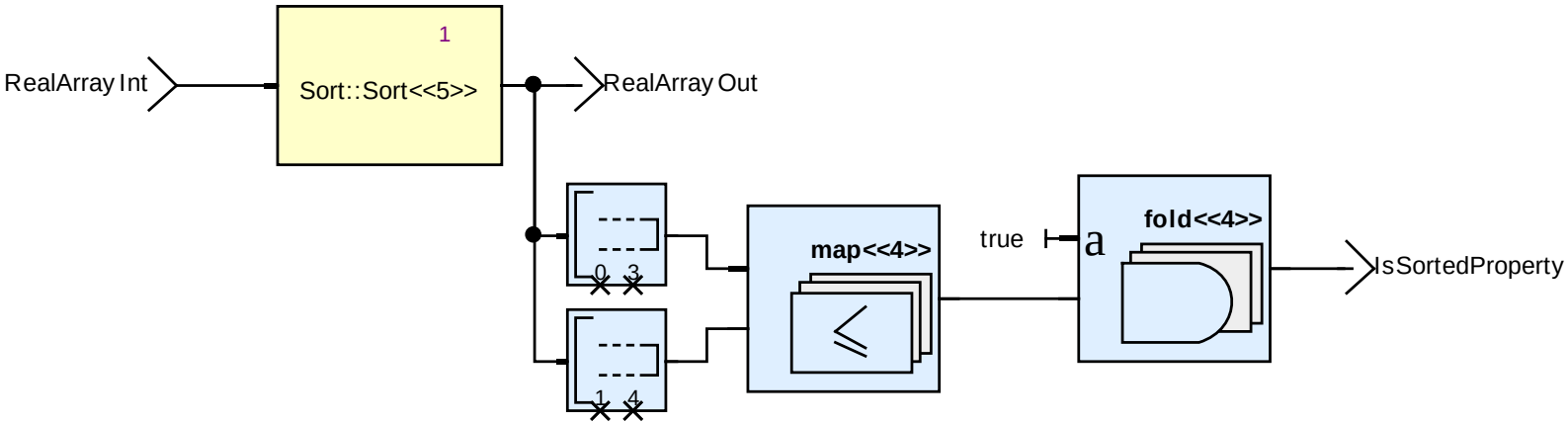
Exercise



Exercise 3

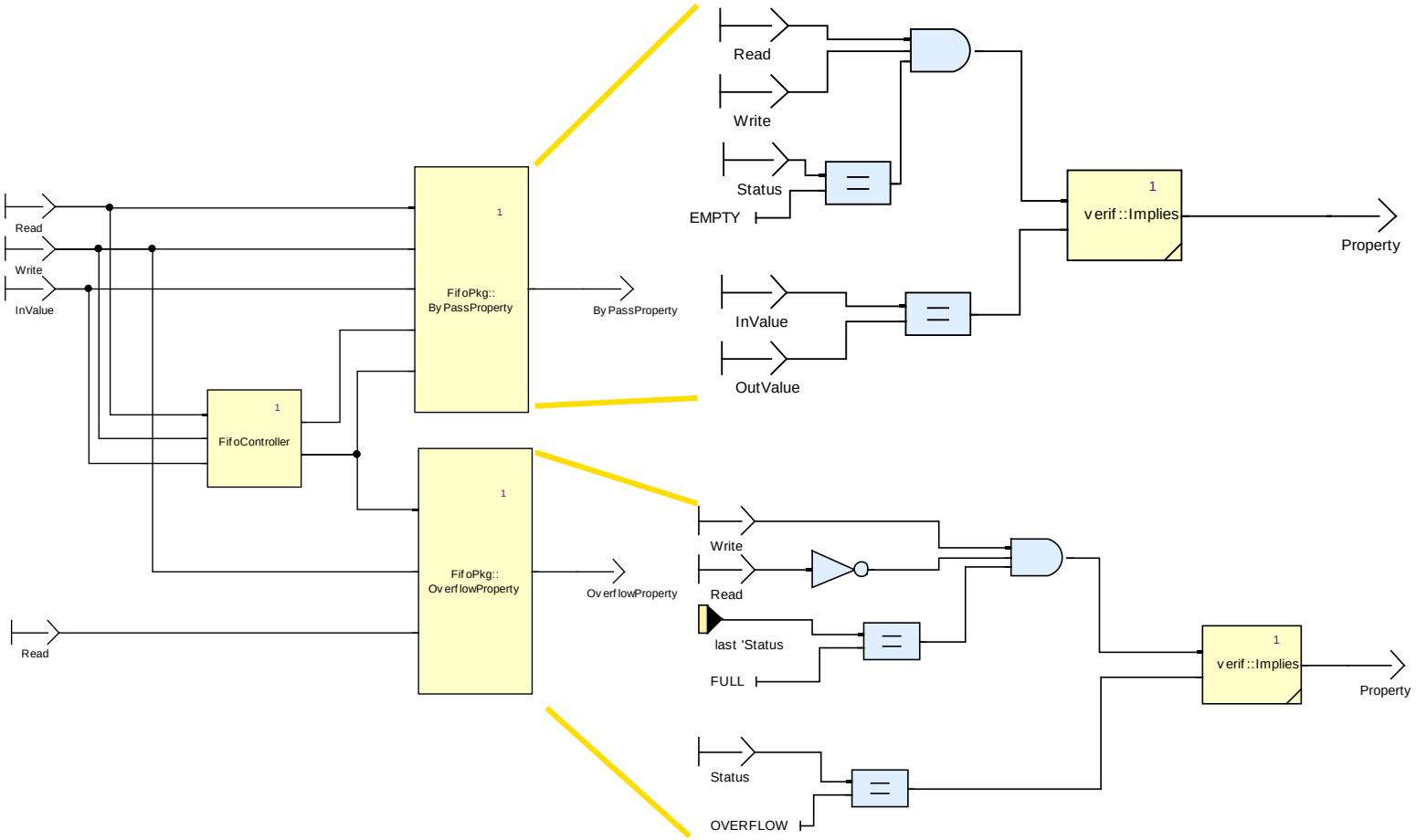


Exercise 4



Exercise 5

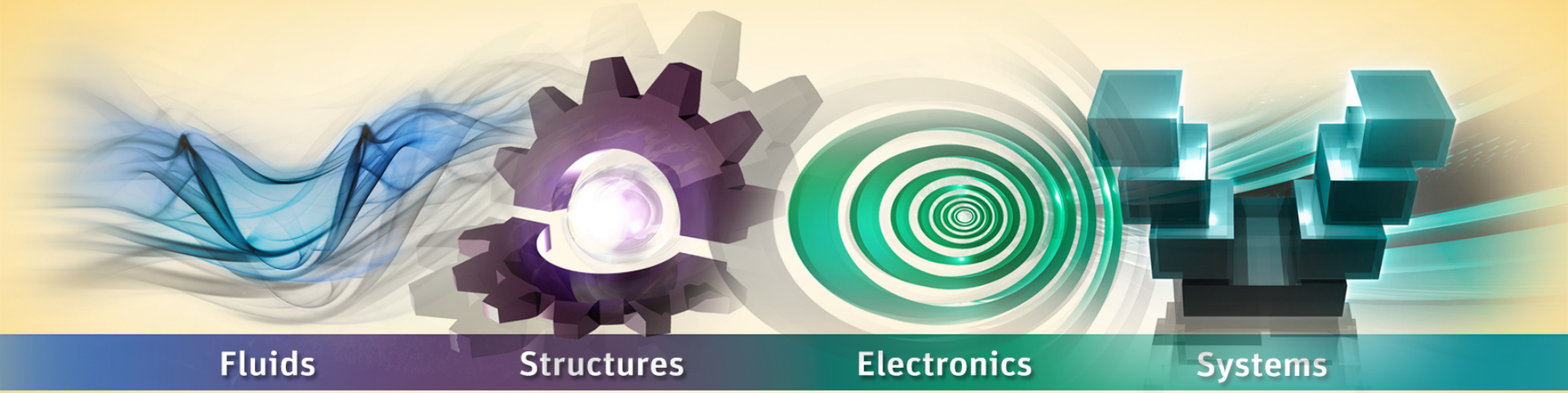
Exercise



ANSYS®

Realize Your Product Promise®

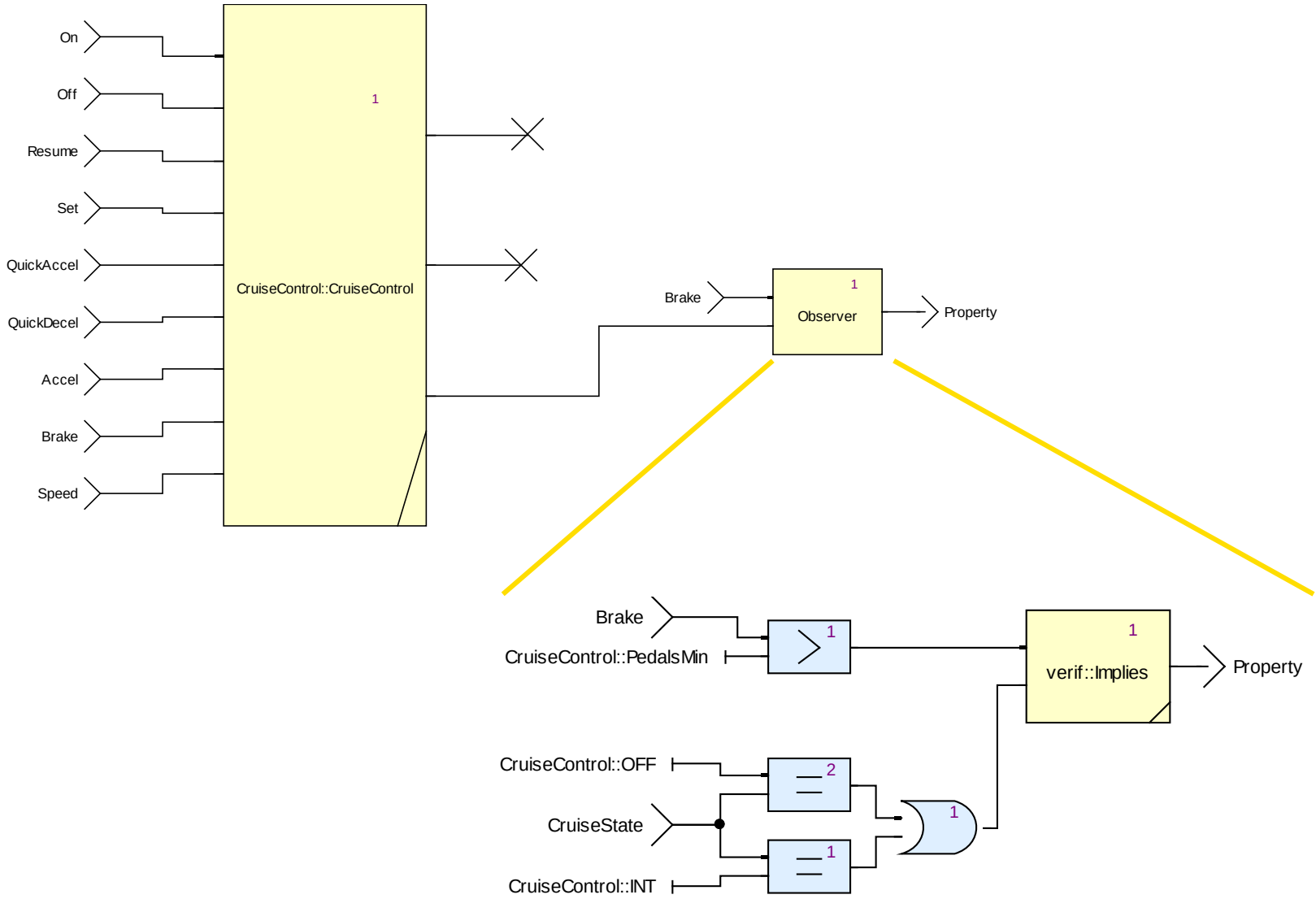
Labs Solutions



ESTEREL Technologies

Case Study: Step 1

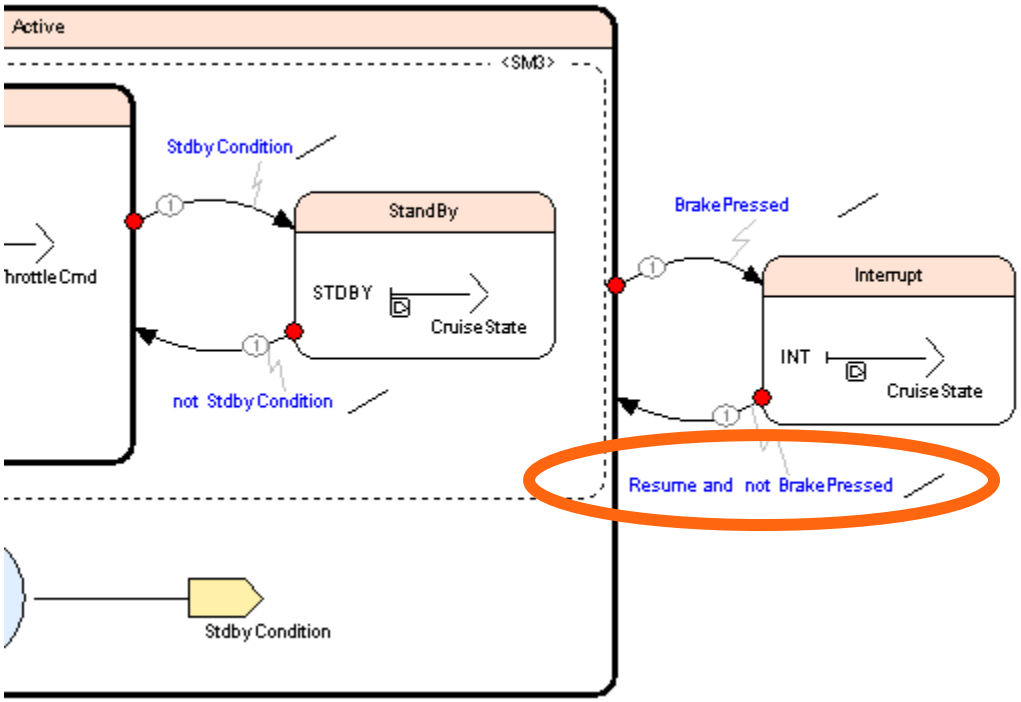
Lab



Case Study: Step 2

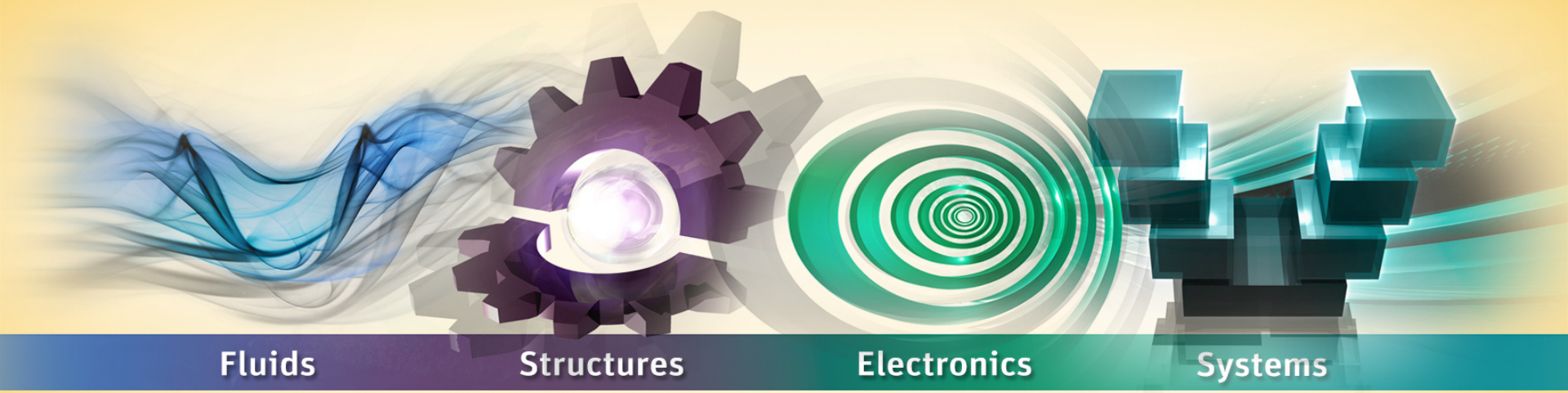
Lab

Cruise Control Fixed



ANSYS®

Realize Your Product Promise®



ESTEREL Technologies