Connecting Requirements to Test Execution with MBSE and MBT

Presented by **Andrew Pollner** and **Katherine Pilat**

Software Engineering Institute
Model-Based Systems Engineering (MBSE) in Practice
August 21, 2025

ALP International Corporation
5425 Wisconsin Avenue
Suite 600
Chevy Chase, MD 20815
(301) 654-9200

Introduction to ALPI

- Provides DoD T&E consulting services (DOT&E, DTE&A, DOD CIO, USAF)
  - Focus on test process improvement (MBSE, MBT)
  - Leverage automation strategies for testing
  - Develop test automation frameworks
  - Deliver accredited ISTQB certification training (AI Testing, Model-Based Testing)
  - Development, Implementation, and Training of DevSecOps tools
  - Support T&E policy and guidance through best practices and whitepapers
- Formed in 1993
- Locations: DC-Metro and Greater Boston Area

Software Testing Challenges

Understanding the Domain

Ensuring Rigor in Test Design

Ensuring Repeatability in Test Execution

Understanding Model-Based Testing (MBT)

- What is MBT?
  - An approach to testing that leverages MBSE to generate test artifacts and improve program integrity through enhanced test design and test execution
- Why Use MBT?
  - Provides comprehensive and efficient verification and validation of a system
  - Enables traceability of requirements through the digital thread from system to test
    - ► Identifies gaps in requirements and coverage
  - Establishes repeatability in testing and reduces maintenance costs

Elements of Model-Based Testing (MBT)

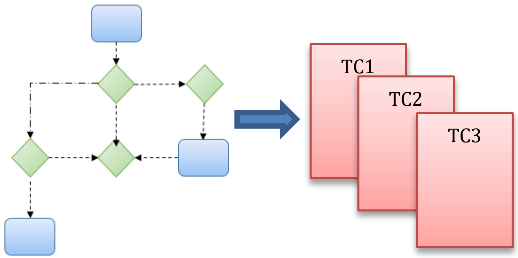
- Three elements of MBT:
  - Model
  - Automated Test Design/Generation
  - Automated Test Execution
- Programs can use any or all elements
  - Implementation can be in stages depending on program maturity
    - New programs may implement the process starting from the System model
    - Existing programs in production can start with Test Model and Test Design

Models

- System Models represent the system, informed by Stakeholder Inputs, Customer Objectives, Mission Capabilities, Mission Requirements, Trade Studies, etc.
  - Describes the internal workings of the system
- Test Models represent the system from the test perspective
  - Contains inputs for testing, reuses system model elements
  - Contains Verification Architecture and V&V information (e.g., test approaches, equipment, etc.)
- Contains Requirements linked to activities, interfaces, use cases, etc.

Automated Test Design

- Generate Test Cases from the Test Model
  - Using Automated Test Design Tools, a model is evaluated and test cases are generated
  - Generated based on criteria, such as “full requirements coverage” or “least amount of test cases required to hit all model paths”
  - Provides Continuous Testing and Decision Point Evaluation

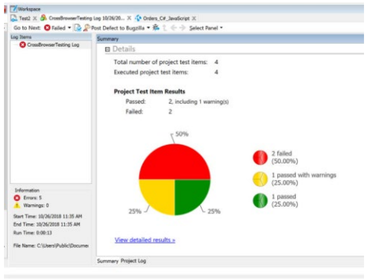


<u>Test Design Example Tools:</u>

- Yest
- Conformiq Designer
- MBTSuite
- Etc.

Automated Test Execution

- Execute Test Cases
  - Test Execution Tools take manual or automatically generated test cases and execute them, providing data and pass/fail results
  - Execute on Models for a baseline expectation of results
  - Execute on System and compare to baseline



Test Execution Example Tools

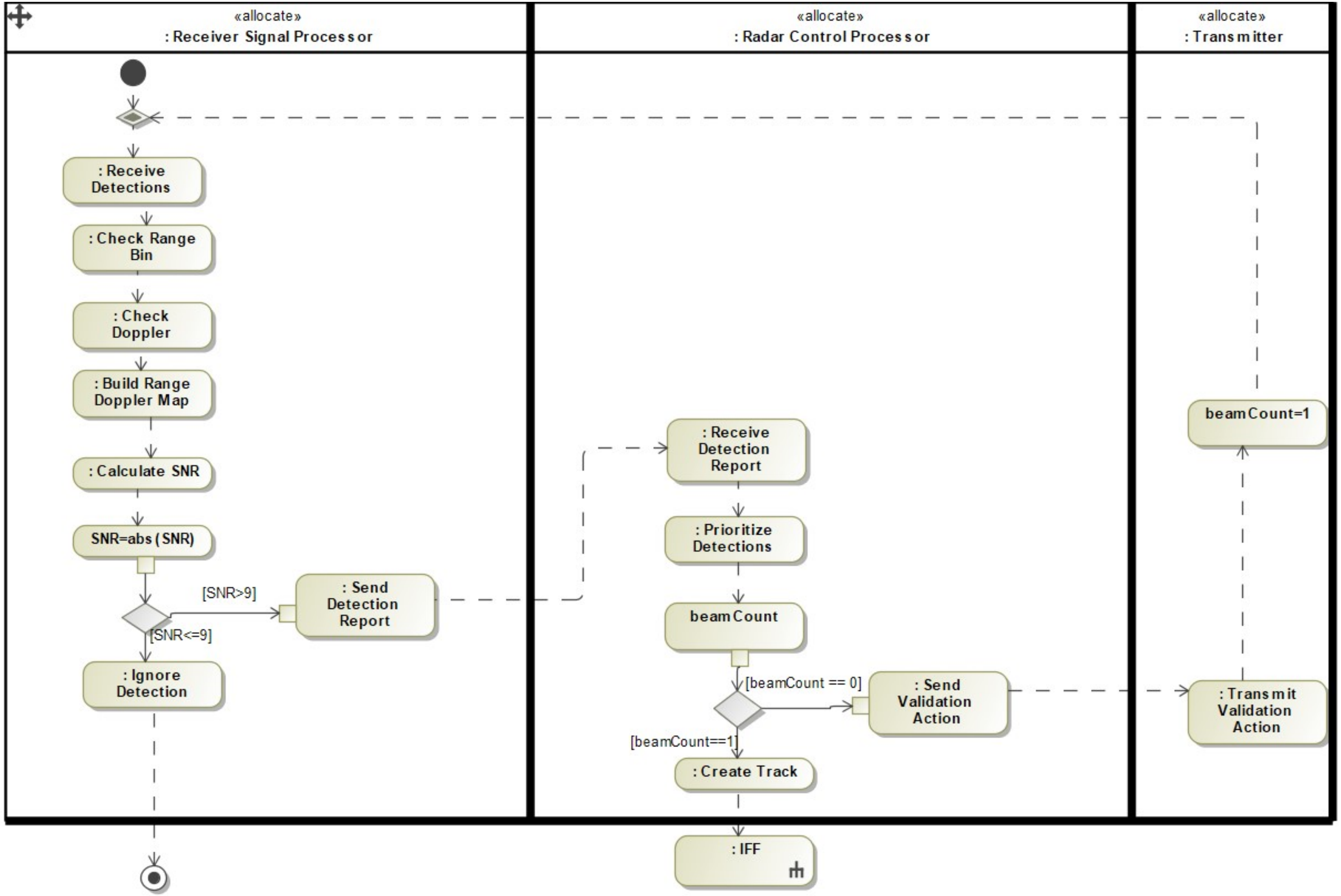
- Conformiq Transformer
- Cameo Systems Modeler + Simulation Toolkit + Jenkins
- Tosca
- Selenium
- Etc.

MBT Prototype

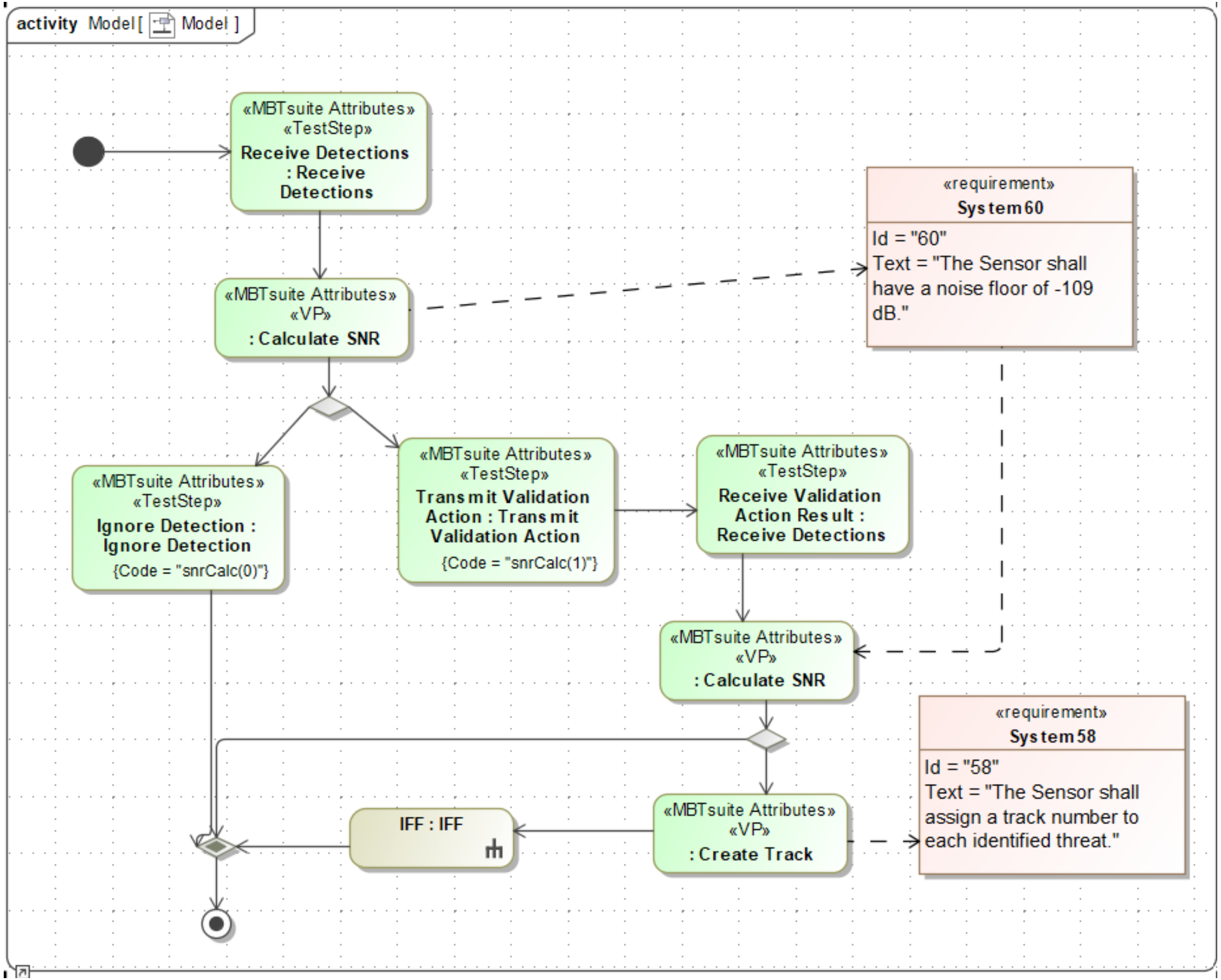
Prototype: Generic Radar

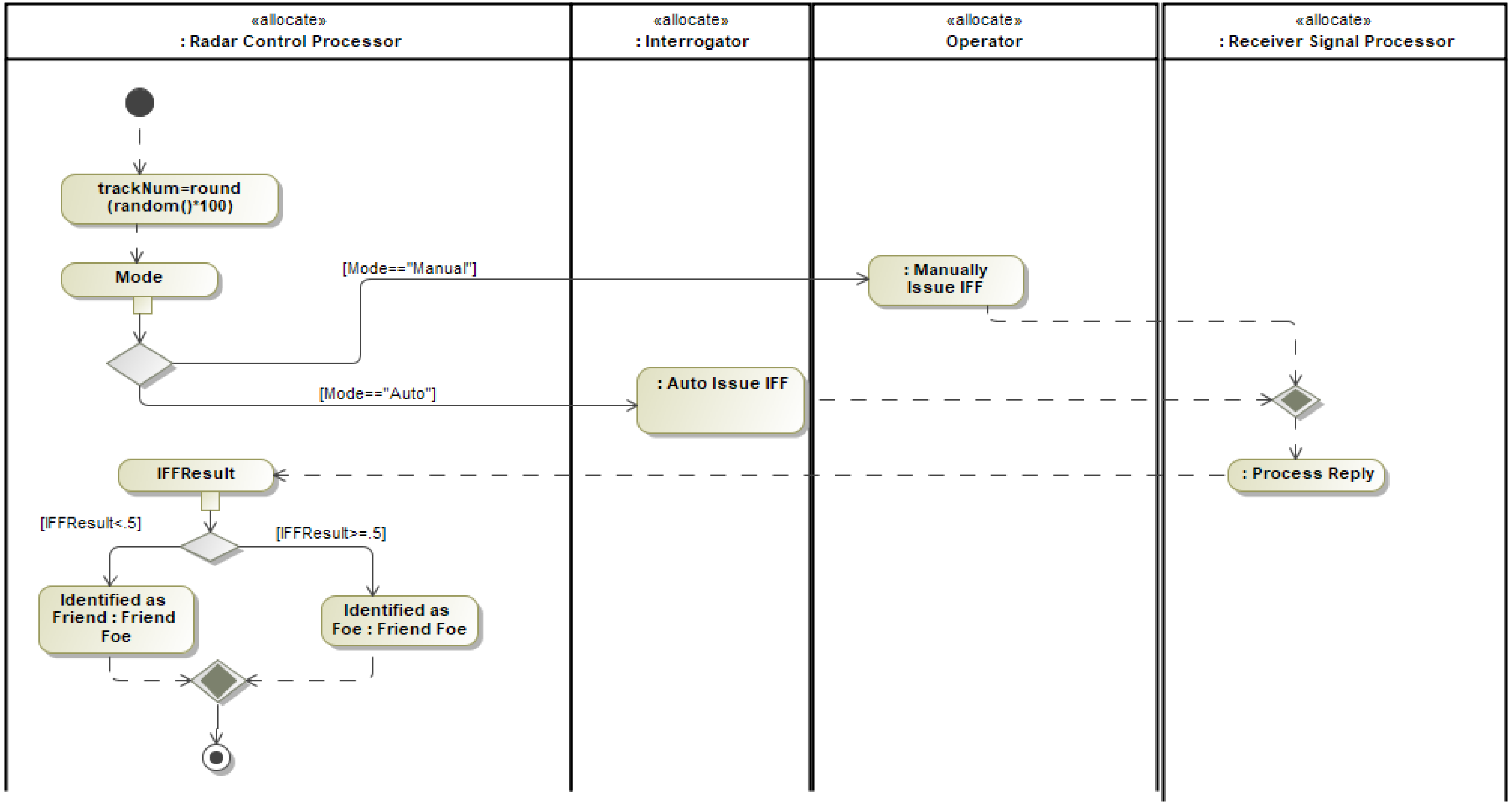
- Use Cases: (1) Search and (2) Interrogation
- System Model
  - One detailed activity diagram for each Use Case
  - Linked to requirements
- Test Model
  - One higher level activity diagram for each Use Case
  - Reuses elements from System Model
  - Linked to requirements
  - Function calls to the tactically representative code
  - Basis for the Test Cases
- Tactically Representative Code
  - Python code of the Use Cases that is executed by Model-Based Testing

Search Use Case
System Model



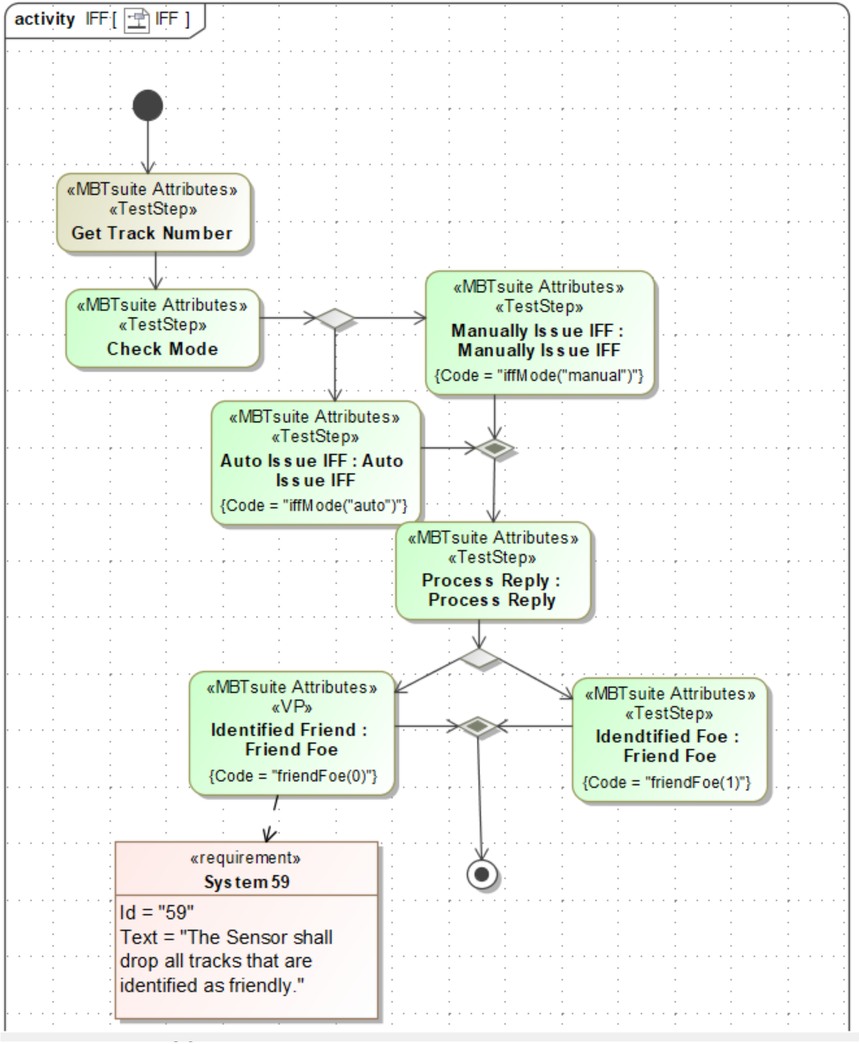
Search Use Case Test Model

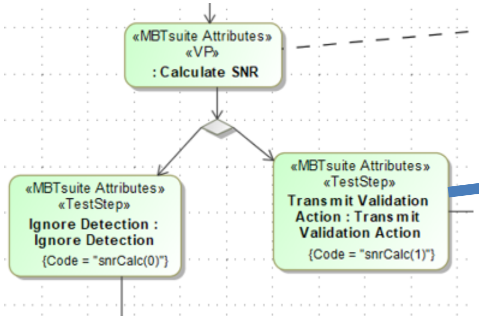


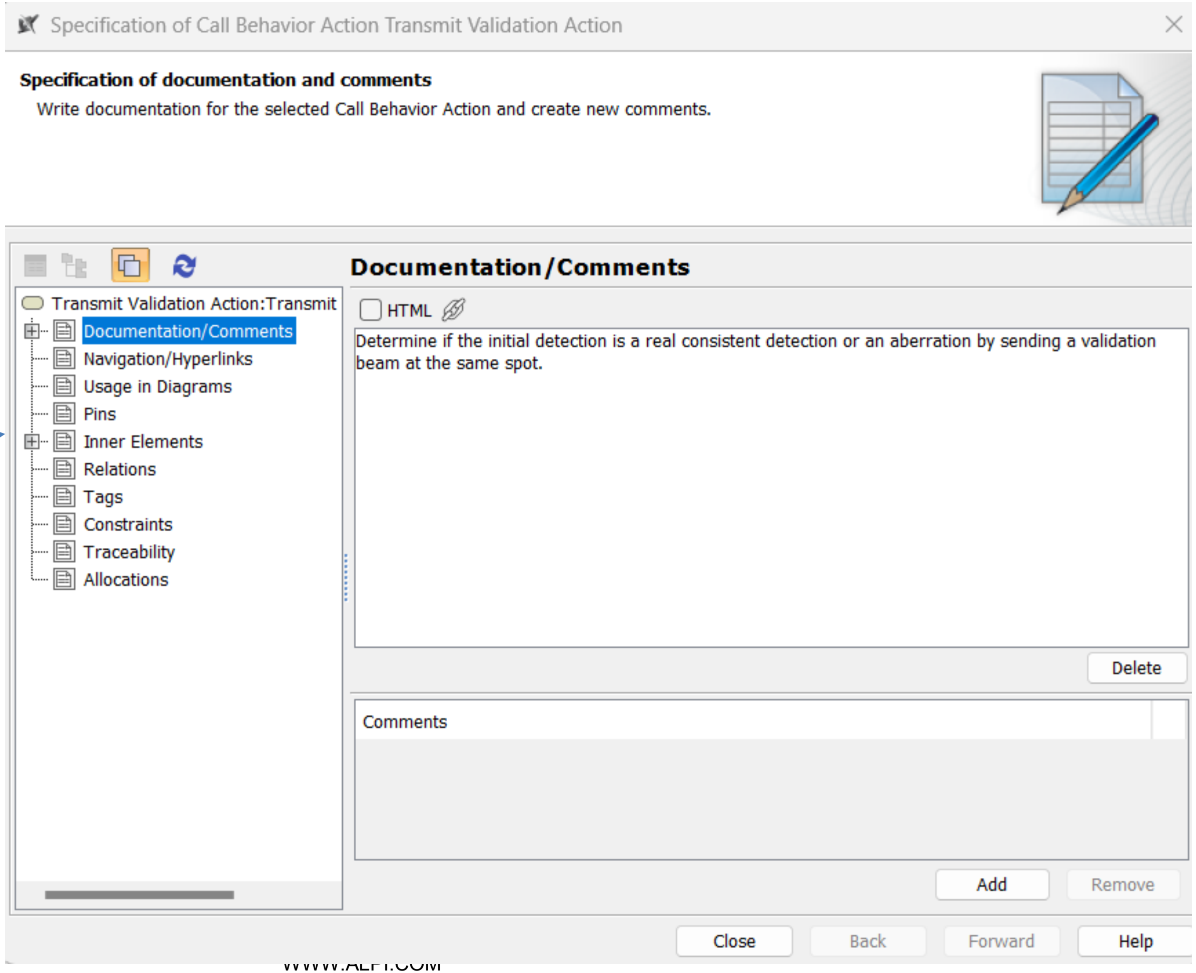


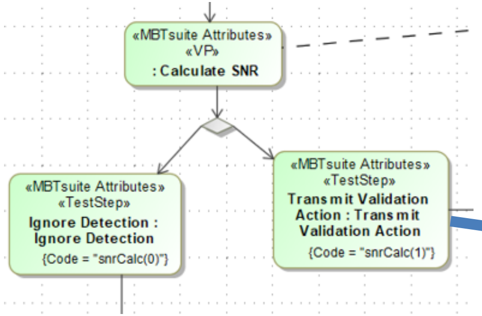
Interrogation Use Case – System Model

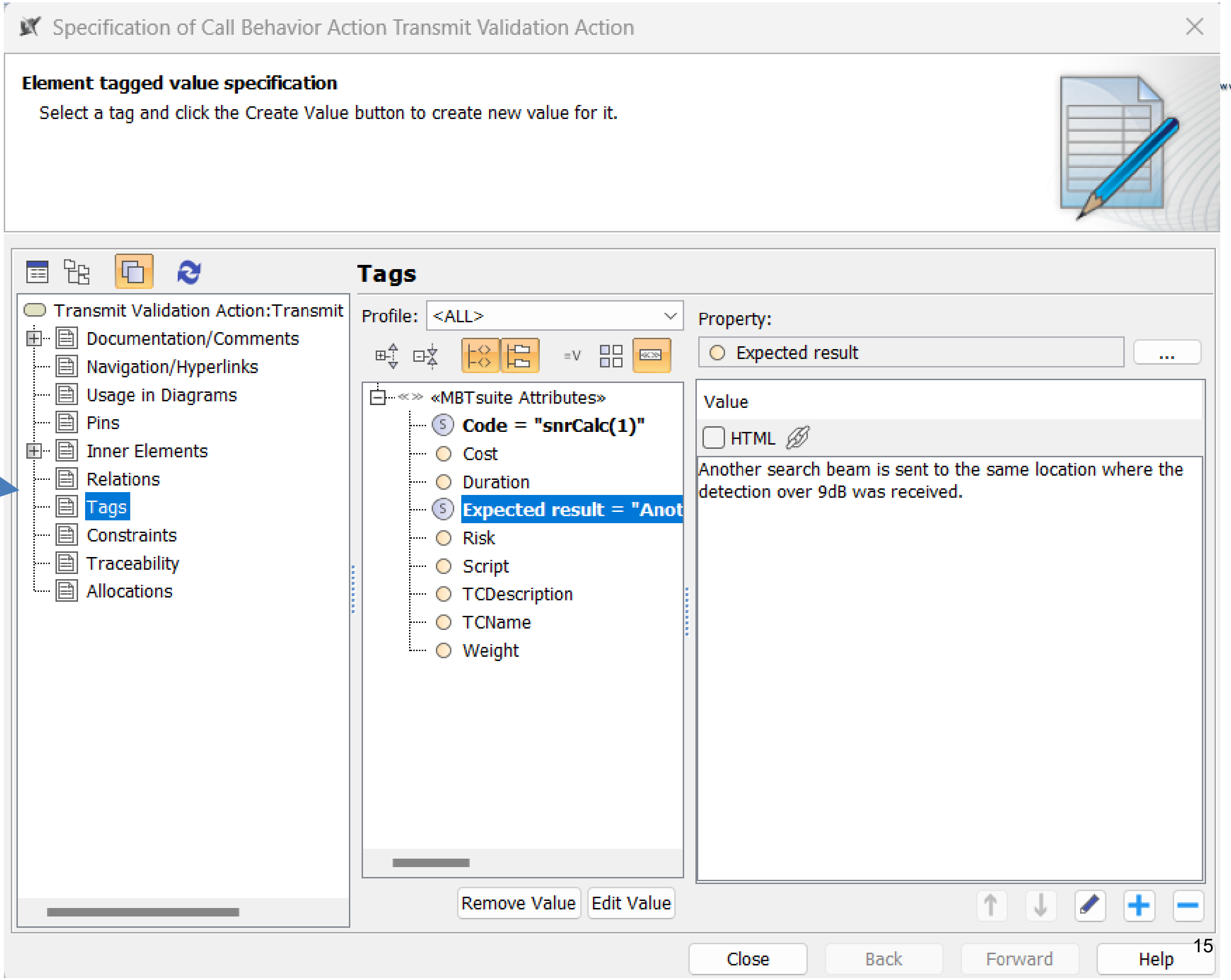
Interrogation Use Case Test Model




«MBTsuite Attributes»
«VP»
: Calculate SNR
«MBTsuite Attributes»
«TestStep»
Ignore Detection :
Ignore Detection
{Code = "snrCalc(0)"}
«MBTsuite Attributes»
«TestStep»
Transmit Validation
Action : Transmit
Validation Action
{Code = "snrCalc(1)"}


Specification of Call Behavior Action Transmit Validation Action
Specification of documentation and comments
Write documentation for the selected Call Behavior Action and create new comments.
Documentation/Comments
Transmit Validation Action:Transmit
Documentation/Comments
Navigation/Hyperlinks
Usage in Diagrams
Pins
Inner Elements
Relations
Tags
Constraints
Traceability
Allocations
HTML
Determine if the initial detection is a real consistent detection or an aberration by sending a validation beam at the same spot.
Delete
Comments
Add
Remove
Close
Back
Forward
Help

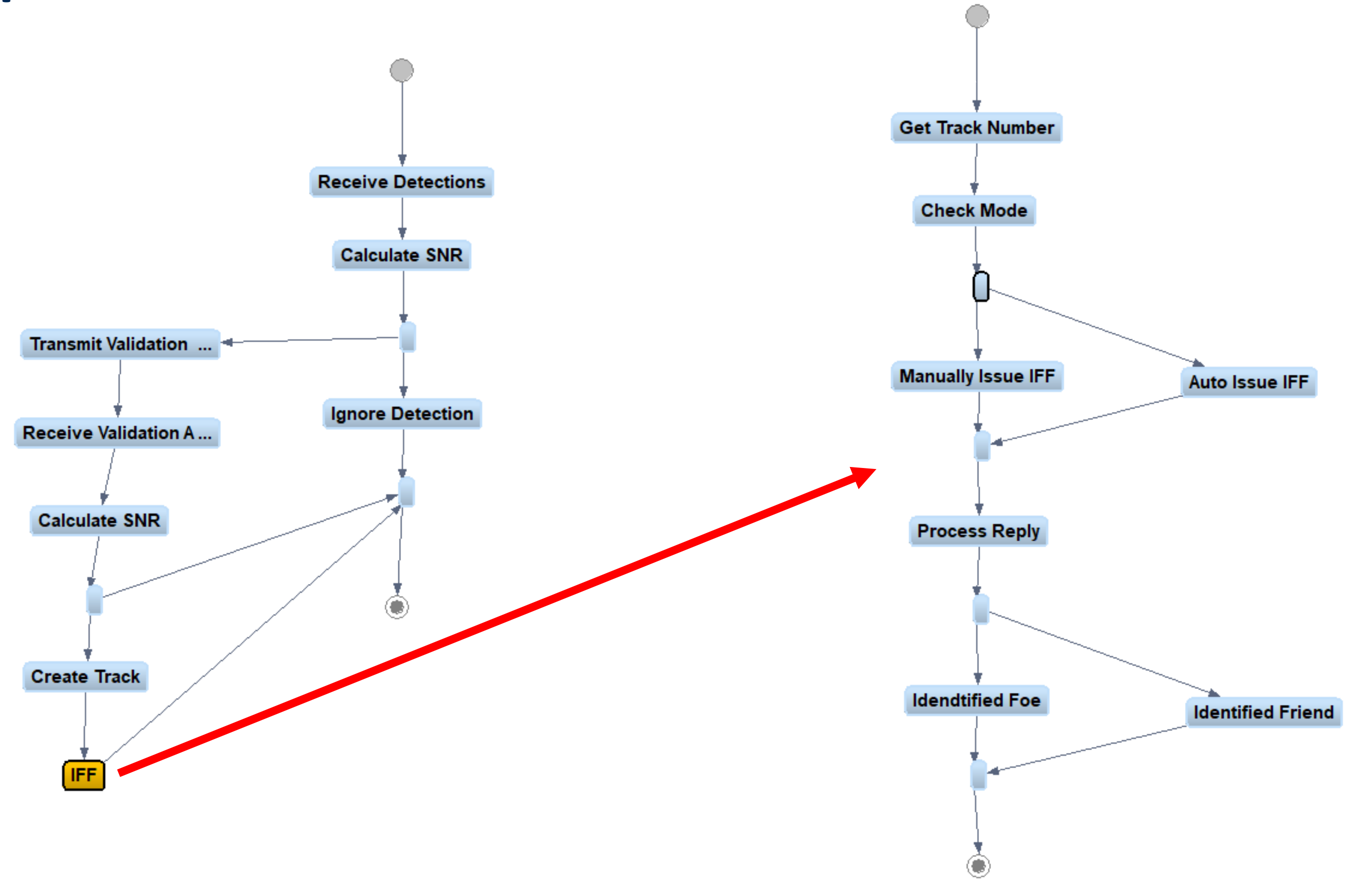

«MBTsuite Attributes»
«VP»
: Calculate SNR
«MBTsuite Attributes»
«TestStep»
Ignore Detection :
Ignore Detection
{Code = "snrCalc(0)"}
«MBTsuite Attributes»
«TestStep»
Transmit Validation
Action : Transmit
Validation Action
{Code = "snrCalc(1)"}


Specification of Call Behavior Action Transmit Validation Action
Element tagged value specification
Select a tag and click the Create Value button to create new value for it.
Tags
Transmit Validation Action:Transmit
Documentation/Comments
Navigation/Hyperlinks
Usage in Diagrams
Pins
Inner Elements
Relations
Tags
Constraints
Traceability
Allocations
Profile: <ALL>
«MBTsuite Attributes»
Code = "snrCalc(1)"
Cost
Duration
Expected result = "Anot
Risk
Script
TCDescription
TCName
Weight
Property:
Expected result
...
Value
HTML
Another search beam is sent to the same location where the detection over 9dB was received.
Remove Value
Edit Value
Close
Back
Forward
Help

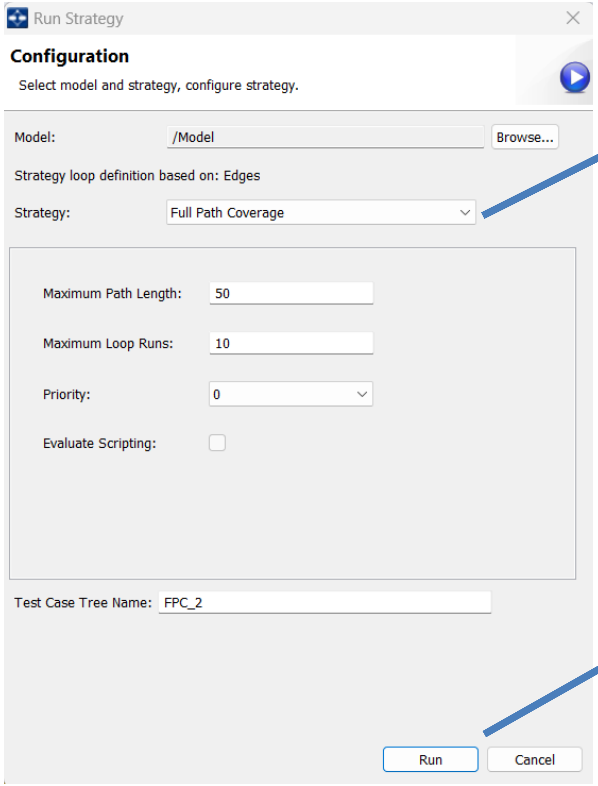
MBT Steps

1. Import Test Model into MBT Tool (e.g., MBTSuite)
2. MBT Tool creates Test Cases based on the paths in the Test Model
3. Export the Test Cases into code (e.g., python)
4. Execute Test Case code to run the tactically representative code

1. Import Model into MBTSuite

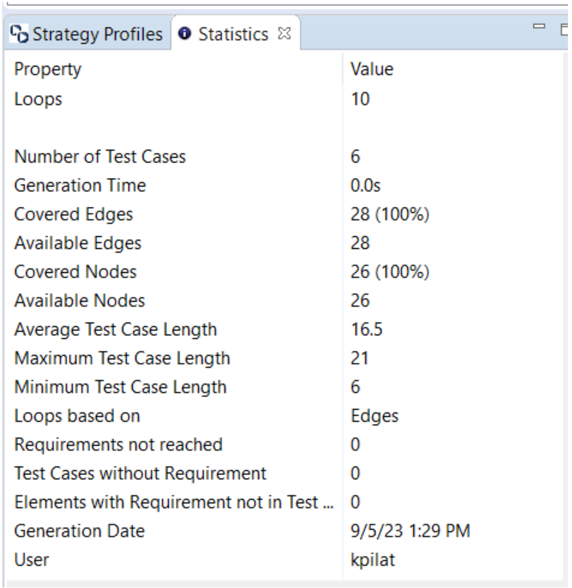


2. MBTSuite creates Test Cases based on paths in Test Model



Different Strategies create different number of Test Cases
- Strategies: Guided Path, Shortest Path, Random

Statistics shown on the created Test Cases



Property	Value
Loops	10
Number of Test Cases	6
Generation Time	0.0s
Covered Edges	28 (100%)
Available Edges	28
Covered Nodes	26 (100%)
Available Nodes	26
Average Test Case Length	16.5
Maximum Test Case Length	21
Minimum Test Case Length	6
Loops based on	Edges
Requirements not reached	0
Test Cases without Requirement	0
Elements with Requirement not in Test ...	0
Generation Date	9/5/23 1:29 PM
User	kpilat

Exportable Test Case in MBTSuite

Test Case #1 – Ignore Detection

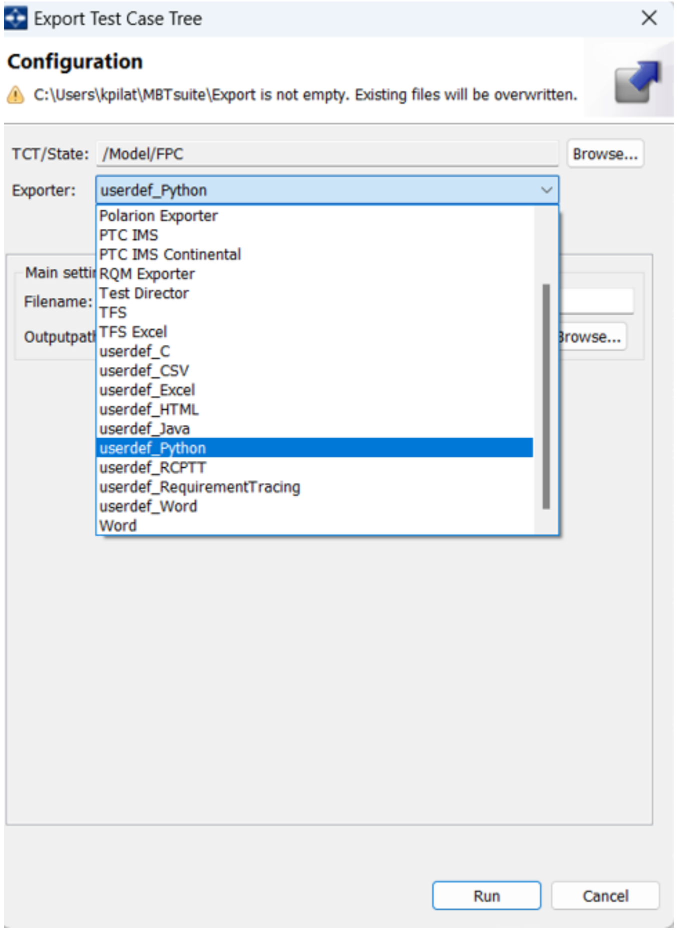
Test Steps:

Step	Type	Step Name	Step Description	Expected Result	Requirements	Code
1	TESTSTEP	Receive Detections	The Sensor receives the detections	The sensor receives 1 or more detections		
2	VP	Calculate SNR	The sensor calcuates the SNR for each detection: Minimum amount of energy we can receive = kTB. Noise Figure = Constant noise from antenna. Radar's Noise Floor = kTB + (10 log (Bandwidth of Receiver)) + Noise Figure.	The SNR calculation for each detection is either >9 or <9	System60	
3	TESTSTEP	Ignore Detection	The detection was not detectable through the constant noise.	The detection was not above the noise floor.		snrCalc(0)

Test Case #2
- Valid Detection
- Manual Mode
- Foe

Step	Type	Step Name	Step Description	Expected Result	Requirements	Code
1	TESTSTEP	Receive Detections	The Sensor receives the detections	The sensor receives 1 or more detections		
2	VP	Calculate SNR	The sensor calcuates the SNR for each detection: Minimum amount of energy we can receive = kTB. Noise Figure = Constant noise from antenna. Radar's Noise Floor = kTB + (10 log (Bandwidth of Receiver)) + Noise Figure.	The SNR calculation for each detection is either >9 or <9	System60	
3	TESTSTEP	Transmit Validation Action	Determine if the initial detection is a real consistent detection or an aberration by sending a validation beam at the same spot.	Another search beam is sent to the same location where the detection over 9dB was received.		snrCalc(1)
4	TESTSTEP	Receive Validation Action Result	The sensor receives the detections back from the validation action.	Result is either another detection or not		
5	VP	Calculate SNR	Calculate the SNR again from the information from the validation action to determine if the siganl reaches above the noise floor again.	The SNR for the detection is either >=9 or <9	System60	
6	VP	Create Track	Put the detection under track and add it to the track list.	Detection is placed under track and added to the track list.	System58	
7	TESTSTEP	Get Track Number	Retrieve the track number from the track list.	Track is retrieved from the track list.		
8	TESTSTEP	Check Mode	Check whether the mode is IFF or Manual	The mode is either automatic or manual		
9	TESTSTEP	Manually Issue IFF	The operator sends an interrogation to the track	The operator interrogates a track.		iffMode("manual")
10	TESTSTEP	Process Reply	Determine whether the reply from the track is a friend or foe based on the code that is returned.	The reply is either a matching code or unmatching/not returned		
11	TESTSTEP	Idendtified Foe	The track's return codes did not match the interrogation codes, or the track did not return anything, so it is classified as a Foe.	The returned code does not match the radar's expected code, or there is no return code.		friendFoe(1)

3. Export the Test Cases into code (e.g., python)



Test Cases Translated into Python

```python
# do your imports here
from RadarCode import snrCalc
from RadarCode import iffMode
from RadarCode import friendFoe

# TestCase_1: Model
# Description:
# Requirements:
#  - System60
print("Model - TestCase_1")
    #The detection was not detectable through the constant noise.
snrCalc(0)
    # Expected Result: The detection was not above the noise floor.

# TestCase_2: Model_0002
# Requirements:
#  - System60, System58
print("Model_0002 - TestCase_2")
# Determine if the initial detection is a real consistent detection or an aberration by sending a validation beam at the same spot.
snrCalc(1)
# Expected Result: Another search beam is sent to the same location where the detection over 9dB was received.

#The operator sends an interrogation to the track
iffMode("manual")
# Expected Result: The operator interrogates a track.

#The track's return codes did not match the interrogation codes, or the track did not return anything, so it is classified as a Foe.
friendFoe(1)
# Expected Result: The returned code does not match the radar's expected code, or there is no return code.
```

Test Case 1

Test Case 2

4. Tactically Representative Code - Run by the Test Cases

www.alpi.com

```python
def snrCalc(snrCount):
    x = 0
    ktb = -174
    noiseFig = 5
    bandwidth = 60
    global firstPass
    firstPass = firstPass + 0
    if snrCount == 0:
        powerRec = random.randint(101,105)
    elif snrCount == 1:
        powerRec = random.randint(95, 100)
    snr = abs(ktb + bandwidth + noiseFig + powerRec)
    print(snr)
    if snr < 9:
        #ignore detection
        print("Detection ignored")
    else:
        if firstPass == 0:
            print ("Sending validation beam")
            firstPass=firstPass+1
            snrCalc(firstPass)
        elif firstPass == 1:
            print("Create Track") #create track array
            trackList.append(random.randrange(0,50))
            print(trackList)
            x=x+1
            firstPass=0
```

```python
# Determines if the radar is in manual or automatic mode. If manual, a gui will pop up for the
# user to click a button to send an IFF. If it is in auto mode, the code will send the IFF.
def iffMode(mode):
    if mode=="manual":
        tk = Tk()
        frame = Frame(tk, borderwidth=6)
        frame.pack(fill=BOTH, expand=6)

        button = Button(frame, text="Send IFF", command=tk.destroy)
        button.pack(side=BOTTOM)
        tk.mainloop()
        print("IFF Sent Manually")
    elif mode == "auto":
        length = len(trackList)
        print("IFF sent on Track ", trackList[length-1])

# Determines if the track that has been interrogated is a friend or a foe. If it is a friend the track
# will be dropped. If it is a foe, the radar will engage the track.
def friendFoe(ff):
    if ff == 0:
        if not len(trackList)==0:
            trackList.pop()
            print("Friend, track dropped")
            print(trackList)
    elif ff == 1:
        print("Foe, engage")
        trackList.pop()
```

Test Cases Results

OUTPUT DEBUG CONSOLE TERMINAL

```
Model - TestCase_1
6
Detection ignored
Model_0002 - TestCase_2
12
Sending validation beam
9
Create Track
[0]
IFF Sent Manually
Foe, engage
Model_0003 - TestCase_3
9
Sending validation beam
10
Create Track
[36]
IFF Sent Manually
Friend, track dropped
[]
Model_0004 - TestCase_4
12
Sending validation beam
14
Create Track
[25]
```

- Currently manually recorded
- Next step would be automated P/F

Test Case	Test Condition	Result
1	The detection is not above the noise floor, ignore.	Pass
2	There is a real detection, radar is in manual mode, and the detection is a foe	Pass
3	There is a real detection, radar is in manual mode, and the detection is a friend	Pass
4	There is a real detection, radar is in auto mode, and the detection is a foe.	Pass
5	There is a real detection, radar is in auto mode, and the detection is a friend.	Pass
6	The validation check returns a detection that is not above the noise floor.	Pass

Challenges in MBT Implementation

- Tools that can import models vs only create models
- Tools that generate scripts vs automated testing solutions
- Limited availability of STAT (e.g., combinatorial methods)
- ATO in Government cloud environments
- Challenges with non-deterministic models (e.g. generative AI)

Summary and Conclusions

Model-Based Testing:

- Illustrates a value-added approach to improving T&E with MBSE
- Allows for end-to-end testing providing traceability and rigor
- Is achieved with the right set of tools and techniques
- Improves the testing practice through shift left
- Practical solution to bridge theory and program success

Q&A