Integrated Modular Avionics: New Challenges for MBT

Nikolay Pakulin
Senior Researcher, ISPRAS, Moscow
npak@ispras.ru

Agenda

- Avionics testing and certification
- MBT Challenges
- PyTESK: MBT and TTCN3 in pure Python

Integrated Modular Avionics

- Avionics = Electronic systems on-board of aircraft
- IMA - new paradigm of avionics
- Before IMA:
  - Each unit has individual control in the cockpit
  - Connected by a dedicated wire;
  - Continuous signals
  - Tons of wires onboard
- IMA:
  - Units are connected to a common bus (AFDX);
  - Discrete signals;
  - Control protocols
  - Readings are processed and displayed by onboard computing nodes
- New avionics options - growth of complexity
  - More operations automated
  - More interconnections between functions

Testing Avionics: Peculiarities

- Really, really high cost of mistake: severe consequences of defects
  - Need for thorough testing
  - Need for **responsible** testing
- Certification of components
  - Testing of artifacts
  - Certified/qualified tools
  - Requirements to processes

Avionics: Requirements to Testing

- Clear and concise tests *to be reviewed(!)*
  - Test procedures consist of steps
  - No ambiguity
  - Test are reproducible
  - No "magic behind curtains"
- Test steps
  - Act, Verify, Ensure
  - PASS, FAIL, ERROR
- Responsible testing: someone must take responsibility for verdict

Avionics Test Procedure: Sample

Test Action	Verdict
1. Push button “Start”	Pass
2. Ensure that indicator “Started” is FLASHING	Pass
3. Wait for 5 seconds; validate that the indicator "Started" is GREEN	Pass
and so on ...	

- The sample for system testing is fictitious
  - But realistic
- Test procedures are presented as MS Word/Excel documents
- Test is "executed" by a human performing the prescribed actions and putting down the verdict
- The human signs the test execution log to take the responsibility for its result

IMA Modeling Challenges

- Need to be close to engineering intuition
  - Continuous time, cooperating active units, numerical modeling
- Need to match "digital" architecture
  - Discrete time messages, integer number limits, floating point precision, scheduling and networking issues
- Models of different systems are not independent
  - Systems are embedded on a plane

Avionics MBT Challenges

- On-the-fly testing
  - Predictable test sequence generation strategy
    - Similar test sequences for similar implementations
    - Well-defined coverage criterion
    - Coverage metrics
  - Clear semantics of test generation
    - Clear test specification: user-defined hints, targets for generation (specific states/transitions to pass, etc.)
    - Comprehensible test data construction: user-defined hints, *solvers*, ~~random walk~~
  - Result review or qualified generation tools
  - Detailed logging, suitable for certification

Avionics Testing Challenges (2)

- Offline test generation
  - Comprehensible test suite for **reliable (i.e. manual)** validation
    - Reasonable number of tests cases
    - Reasonable number of test steps per test case
    - Clear and explicit test purposes generated
    - Requirements tracking for generated test cases
    - Predictable generation strategy
    - Detailed logging, suitable as evidences for certification
  - Or qualified generation tools

MBT Notation: Domain-specific vs Special-purpose vs General Languages

`	Domain	Special	General
Learning curve **or** specialists available in the market			
Domain-specific adaptation			
Portability to other domains			
Variety of tools in the market			
Ease of analysis and processing for test generation			
Ease of test review			
Ease of qualification			
Ease of share with partners			

- Domain specific languages model system in terms close to engineering concepts of the domain
- Special-purpose language provides constructs for modeling (e.g.SDL) or testing concepts (TTCN3)
- General-purpose languages: C/C++, Java, C#, Python, etc.

Easy	Not so easy	Complicated	Tough

ISPRAS Ongoing Research: PyTESK - MBT in Python

- Why Python:
  - Simple, feature-rich, extensible
  - Large codebase / bindings to powerful libs
  - Rich set of tools and toolkits
    - Debugger!
    - Eclipse integration
- PyTESK planned to release Dec. 2012
  - Components, message and *sampling* ports
  - On-the-fly and *offline* test generation
  - Altsteps in test cases
  - Contract specifications and *state machine models*
  - *AADL integration*

PyTESK Origins

- MBT framework UniTESK
  - Contract specifications: pre- and post- conditions
  - On-the-fly test generation
- SDL
  - Message channels
  - State machines
- TTCN3
  - Components, ports
  - Alt-statements, templates, timers

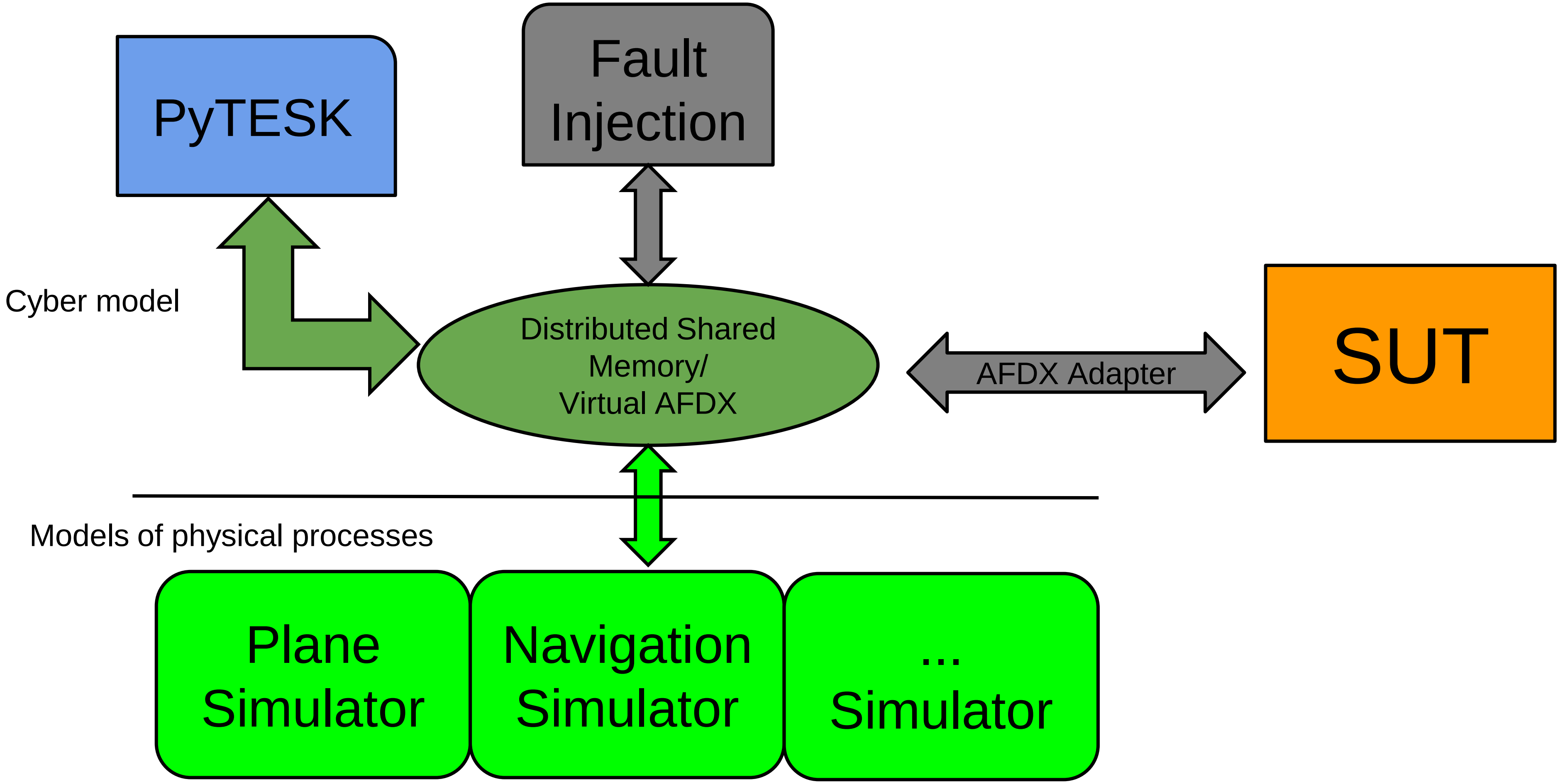
PyTESK Look-n-Feel

```
@AltStep
def waitForAck(port, expectedAckNumber, timeout):
    @OnMessage(port, Template(type = tftp.Ack, blockNumber = expectedAckNumber))
    def ack_received(msg):  return msg

    @OnMessage(port, Template(type = tftp.Ack))
    def data_with_wrong_blockNumber(msg):  Assert.fail("Received invalid ACK,")

    @OnMessage(port, Template(type = tftp.Error))
    def error(msg): Assert.inconclusive("Error message received: %s" % msg)

    @OnMessage(port)
    def any_message(msg): Assert.fail("Unexpected message: %s" % msg)



    @OnTimer(timeout)
    def timeout():  Assert.fail("Timeout: expected ACK with block number %d"%expectedAckNumber)
```

PyTESK vs TTCN3

- PyTESK is based on TTCN3 runtime semantics
- Python has simpler grammar, simple extension mechanisms, object-oriented paradigm, rich standard library, powerful open tools
- Python misses static typing system
  - Partially supported by PyDev environment

PyTESK IMA Test Harness



Conclusion

- IMA are cyber-physical systems and demand new approaches to modeling
- High cost of fault imposes strict requirements on test design and execution
- The industry is conservative and sticks to "proven" methods
- Tools qualification required