Acquisition Research Program:
Creating Synergy for Informed Change

Use of Automated Testing to Facilitate Affordable Design of Military Systems

V. Berzins, P. Van Benthem, C. Johnson, B. Womble

The views presented is this paper are those of the authors
and do not necessarily represent the views of DoD or its Components.

The Naval Testing Challenge – Infinity is a Big Place

- Size and Complexity
  - The environment is harsh and boundless
  - There are hundreds of systems on warships
  - Huge state spaces and many system configurations
- Risks
  - Lives are at stake
    - Participants rely on simultaneous correct execution
    - All systems/variants must interoperate seamlessly
  - Software vulnerabilities can be deliberately placed
    - Statistically invisible: enormous amount of room to hide back doors
  - Software can be compromised at runtime
    - Injected faults not present in the version under test, need runtime monitoring
- Testing is necessarily sparse relative to the entire space
  - Exhaustive testing is physically and economically impossible

Emerging Solutions

- **Testing Technologies, Processes & Policies**
  - Safely Reduce Testing Required (2007-2014)[1]
  - Make testing more effective
    - Risk-based testing (2012-2014)[1]
    - Safe test result reuse (2009)[1]
  - Reduce System Integration problems
    - Continuous automated end-to-end testing (2015)
    - Architecture QA aimed at system interference (2014)[1]
- **Practices and guidance are being developed**
  - OSA Technical Reference Frameworks (2015)
  - Affordable Design and Test Guidelines (2015)

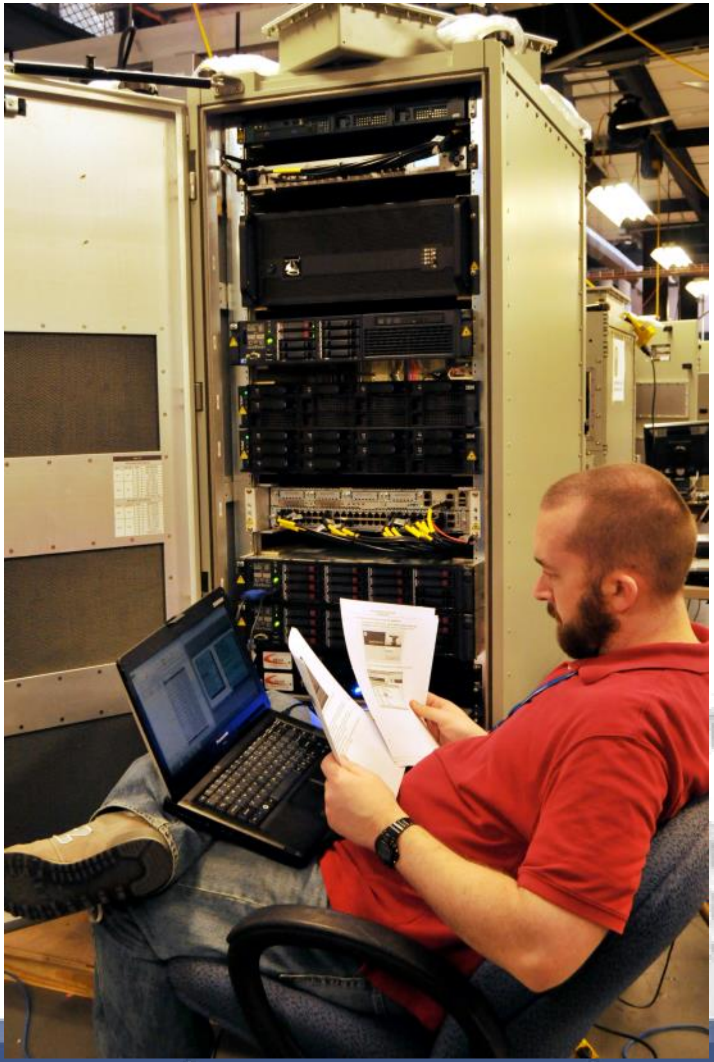
[1]See prior year ARP Symposium Proceedings

Some Bugs are Must Fix

- **Risk analysis to identify critical services**
  - Hazards: what can go wrong
  - Severity: how bad would it be
  - Risk Budget: how often can we accept it
  - Prevent worst hazards: one failure is too much
- **Dependency analysis**
  - To identify internal functions affecting critical services
- **Automated test samples from risk tolerance**
  - Test sets large enough for high confidence
  - Sampling error less than acceptable failure rate

Critical Faults

- **Finding critical faults may require cheating**
  - Statistically invisible = impossible to detect by black box testing
  - Clear box testing can do better
    - Use constraint solvers to synthesize test inputs for majority of cases



Testing is a Design Requirement

	Level	Testability Level Description
0	inadequate	Does not meet requirements for any of the higher levels
1	syntactic	All services and data elements provided by each procurable component have published interfaces/data models that provide names and type signatures.
2	semantic	Published interfaces include precise definitions of the meaning of the services/data, including units, connection to real world objects, and requirements on outputs and final states resulting from all services
3	robust	Published interfaces include all assumptions and restrictions on inputs and states, triggering conditions for all exceptions, and expected results after exceptions
4	observable	All system attributes relevant to checking the requirements are observable either via the published operational interfaces or published augmented testing interfaces
5	measurable	All properties needed to check the requirements have clearly defined measurement and evaluation procedures
6	decidable	Pass/fail decisions for all test cases can be made entirely by automated procedures, without need for subjective human judgment
7	unbounded	Any number of random test inputs can be automatically generated and corresponding test results can be automatically checked for all services

Testing is a Design Requirement

- **QA for architectures should assess their testability levels**
  - Levels 5-7 appropriate for reliable architectures

- **Testability levels 6 and 7 can be augmented with Built-in-Test capabilities**
  - Enables checking system readiness in the field
  - Prognostics: e.g. replace battery soon
  - Reconfiguration: e.g. new load-out for aircraft
  - Device failure: e.g. replace hard drive
  - Corrupted software: e.g. re-image OS

Automation Can Improve Testing

- **Faster development time**
- **Stable and consistent quality systems**
- **Lower costs**
- **Allow fast regression testing**
- **Changes in approach are required**

Testing Infrastructure

- **Programs approach testing differently**
- **Common instrumentation of SW could allow formalization of automated testing**
- **Adopting similar Technical Reference Frameworks enables use of common tools**
- **DoN is considering sponsoring standards for testing**

Hardware Testing

- **Easier than software testing**
  - Uniform state representation
  - Known expected outputs
  - Effective error models

Software Testing

- **More complex failure patterns**
- **Complete test sets not algorithmically computable in the general case**

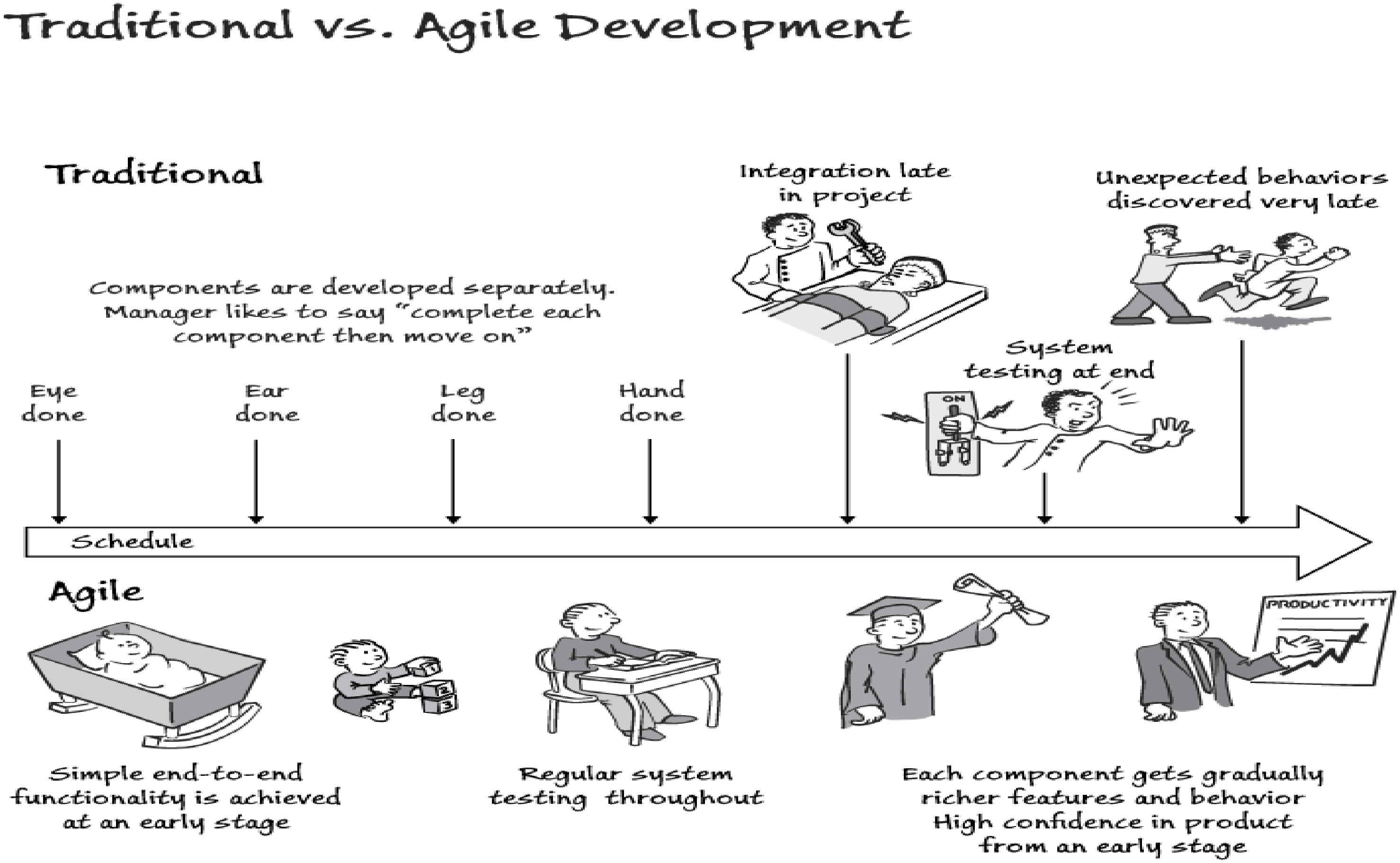
Integration and Architecture Faults

- Hypothesis: many system integration problems are due to architecture faults and imperfections in test and evaluation.
  - Examples of integration problems due to architecture faults are in the backup slides.
  - Testing imperfection example:
    - Code faults in which components fail to conform to architecture standards are missed by test cases.
    - When two components are connected, one triggers such a fault in the other, by exercising an untested situation.
    - Incidence can be reduced by automated statistical testing, with enough test cases for high confidence.

Experience with Automated Testing

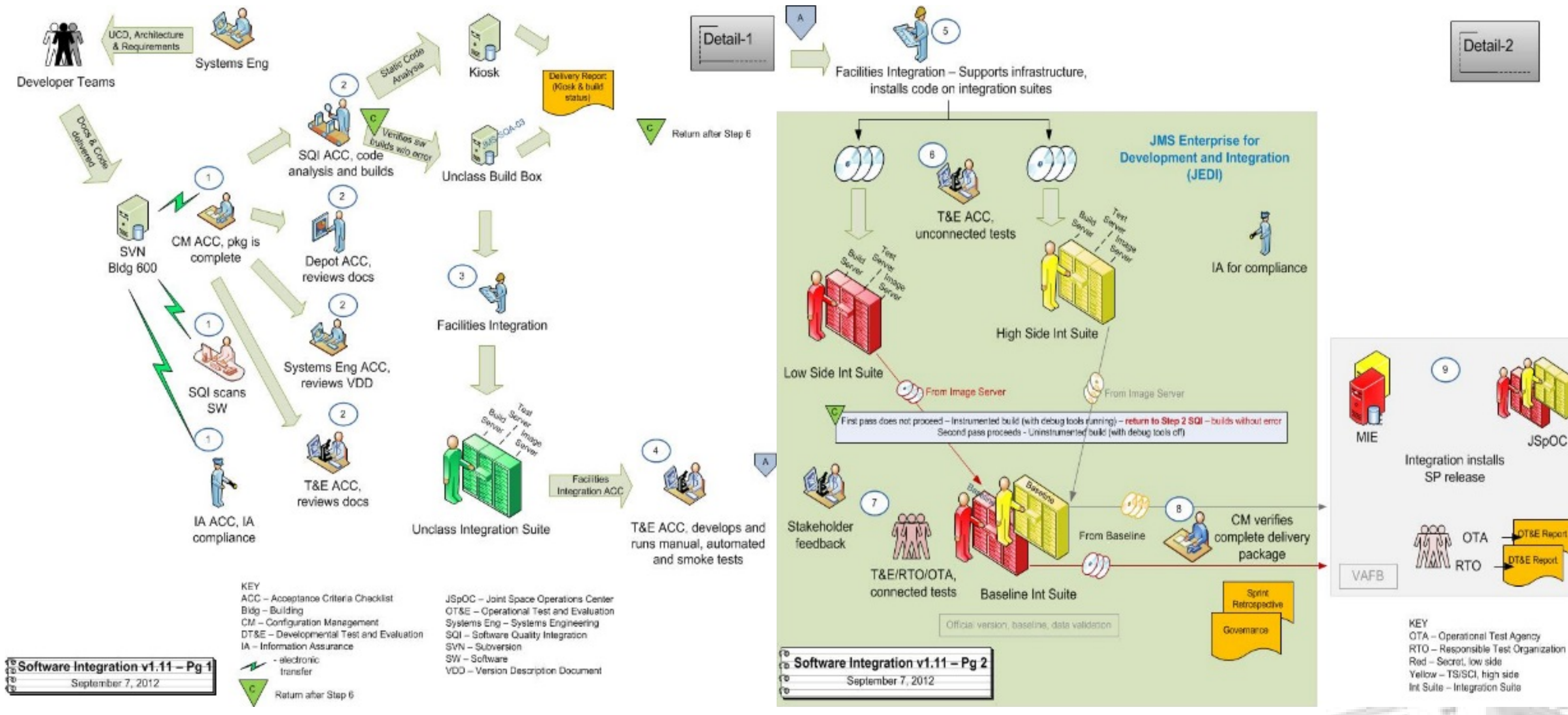
- **Rapid Integration and Test Environment (RITE)**
  - SPAWAR initiative
  - Fundamental change to DoD integration activities
  - Graduated set of tests
- **Focused testing in three phases is a fundamental aspect**
- **Continuous integration process**
- **Black box and clear box testing**

Agile vs Traditional[1]



[1] Slide courtesy of IDT Corp.

RITE Continuous Integration Process



Conclusions

- Clear box testing can expose statistically invisible faults, including malware.
- Incremental development with continuous testing can reduce integration problems.
- System integration problems can be caused by architecture faults.
- QA procedures for architectures should be part of OSA processes.

Recommendations

- Early and continuing automated end-to-end testing should be used to reduce/mitigate system integration risks.

- Source code should be a required deliverable to enable clear box testing and static analysis thereby reducing risk of malware.

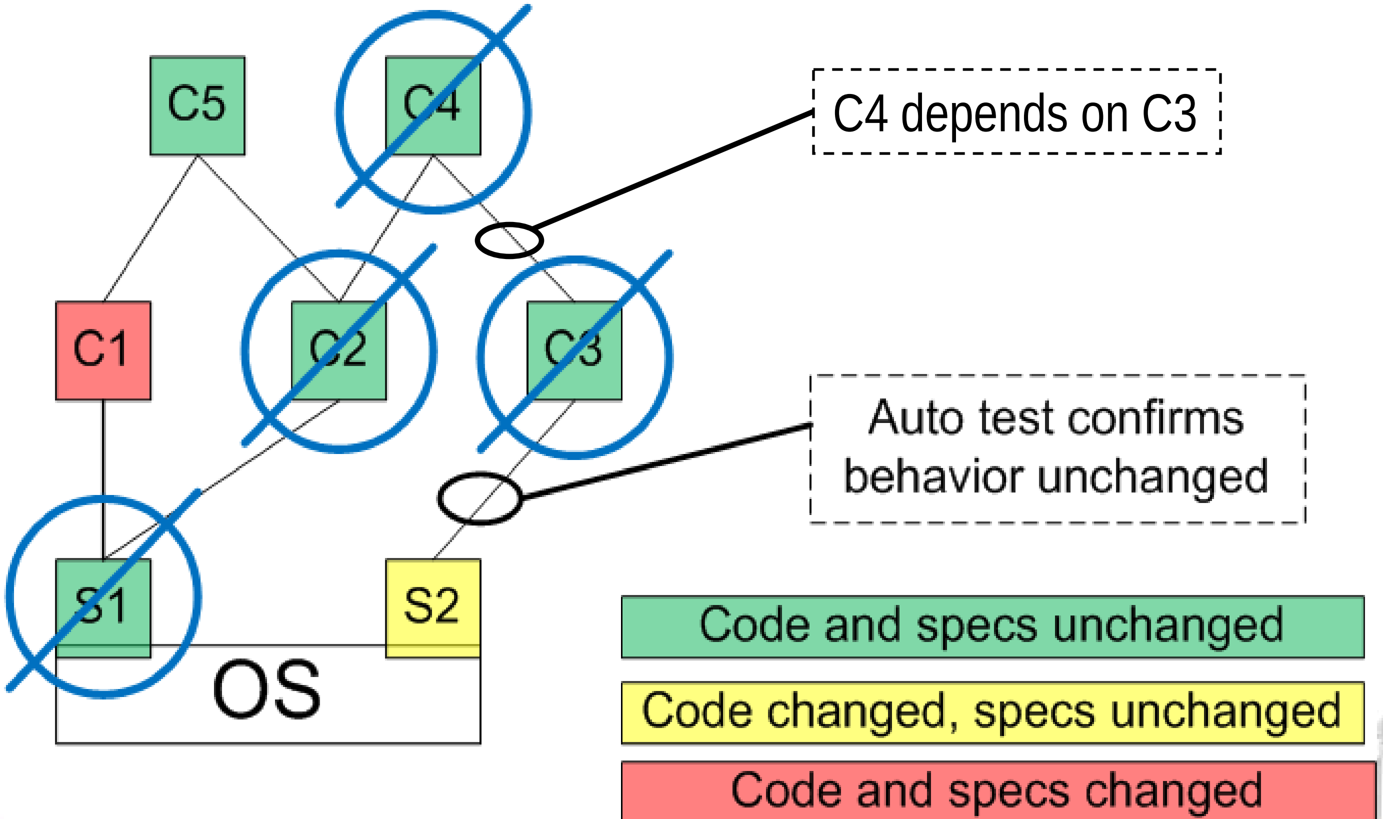
- Runtime infrastructure for detecting/undoing unauthorized changes to code should be part of any OSA/TRF, to reduce cyber risks.

Thank you

Backup Slides

Test Avoidance Approach


= No retest due to slicing and invariance testing



Program Slicing

- Program slicing is a kind of automated dependency analysis
  - Same slice implies same behavior
  - Can be computed for large programs
  - Depends on the source code, language specific
  - Some tools exist, but are not in widespread use
  - No tools spanning boundaries between languages (yet)

- Slicing tools must handle the full programming language(s) correctly to support safe reduction of testing.

Test Reduction Process (1)

- Check that the slice of each service is the same in both versions (automated)
- Check that the requirements and workload of each service are the same in both versions
- Must recheck timing and resource constraints
- Must certify absence of memory corrupting bugs
  - Popular tools exist: Valgrind, Insure++, Coverity, etc.
- Must ensure absence of runtime code modifications due to cyber attacks or physical faults
  - Cannot be detected by testing because modifications are not present in software versions under test
  - Need runtime certification
    - Can be done using cryptographic signatures (Berzins, 2009)

Test Reduction Process (2)

- The test reduction process in the previous slide is for new releases with the same operating environment.
  - This is a significant constraint because reliability depends strongly on operating environment
  - The same system can have 0% reliability in one environment and 100% in another

- Components reused in different contexts need a different approach
  - Can reuse some previous test results and focus new tests on unexplored parts via differences in operational profiles
    - See (Berzins 2009) for details.
  - Risk-based testing can determine number of test cases needed

Risk Based Testing

1. Whole-system operational risk analysis identify potential mishaps / mission failures
2. Identify which software service failures would lead to identified mishaps
3. Use slicing to identify which software modules affect the critical services
4. Associate maximum risk level of affected services with each software module (2012)
5. Set number of test cases using risk level (2008)

Architecture Fault Example (1)

- Kitchen plan calls out a Miele microwave oven and an electric outlet.
  1. Electrical contractor installs a 110 volt outlet.
  2. Oven delivered, installation guide requires a 220 volt power supply, installation fails.

- Architecture left out constraints needed to ensure the subsystems will work together.
  – In this case: power supply voltage.

Architecture Fault Example (2)

- Laundry plan calls out an outlet, water supply, and drain (washer), an outlet, gas supply, and air vent (drier), and a big window on top.
  1. Plumber installs the pipes below the structural members supporting the window.
  2. Electrical contractor finds space for the electrical outlets completely obstructed by the pipes.
- Architecture left out constraints to deconflict resource requirements for the subsystems.
  – In this case: volumes of physical space.