Automated Generation of Adaptive Test Plans for Self-Adaptive Systems

Erik Fredericks and Betty H. C. Cheng
May 19th, 2015
Run-time testing provides assurance for self-adaptive systems (SAS)

An SAS can experience uncertainty, possibly rendering test cases created at design time irrelevant

Proteus ensures that test suites and test cases remain relevant throughout SAS execution
Agenda

- Background
- Proteus approach
- Case study
  - Discussion
- Impact of run-time testing
  - Discussion
- Related work
Remote data mirroring
Goal-oriented requirements engineering
Software testing
Remote Data Mirroring Application

- **RDM** [Veitch2003, Keeton2004] provides:
  - Data protection
  - Prevents data loss and maintains availability
  - Stores data in physically remote locations
  - Represented as an SAS
Remote Data Mirroring Application

Background

Approach

Case Study

Impact

Related Work
Network Connections
Dropped Message
Disrupted Connection
Reconfiguration
Goal-Oriented Requirements Engineering

Partial KAOS [Dardenne1993,vanLamsweerde2009] goal model of RDM
Goal-Oriented Requirements Engineering

Non-invariant

Requirement

(A) Maintain [DataAvailable]

(B) Maintain [Operational Costs ≤ Budget]

(C) Achieve [Minimum Num Links Active]

(D) Achieve [Network Properties]

(E) Achieve [Workload Measured]

(F) Achieve [Network Partitions == 0]

(G) Achieve [Cost Measured]

(H) Achieve [Activity Measured]

(I) Achieve [LossRate Measured]

(J) Achieve [Capacity Measured]

(K) Achieve [Link Deactivated]

(L) Achieve [Link Activated]

(M) Achieve [Link Deactivated]

(N) Achieve [Link Activated]

Agent

Link Sensor

RDM Sensor

Network Actuator
Utility Functions

\[
utility_{Goal_B} = 1.0 - \frac{\text{operational}\_\text{cost}}{\text{budget}}
\]
- **Requirements-based testing**
  - Validate that system under test is satisfying requirements
    - [Myers2011,IEEE2010]

- **Regression testing**
  - Re-validate system following major change to software
    - [Myers2011,IEEE2010]
Terminology

- Test case
  - Single test to assess all or a portion of a requirement

- Test specification
  - Set of all possible test cases derived for a software system

- Test suite
  - Subset of test cases from the test specification
  - Typically derived to be executed under a particular operating context
Software Testing

- Terminology
  - Test Suite 1
    - TC1..TC5
    - TC6
    - TC7
    - TC8
  - Test Suite 2
    - TC1..TC5
    - TC6
    - TC8
    - TC10

- Test Specification
  - TC1
  - TC2
  - TC3
  - TC4
  - TC5
  - TC6
  - TC7
  - TC8
  - TC9
  - TC10
  - ...

- Invariant
- Non-invariant
Proteus Approach

- Requirements-driven approach for managing run-time testing

- Defines an **adaptive test plan** for each SAS configuration
  - Each configuration corresponds to a particular set of environmental conditions, or *operating context*

- Performs a testing cycle during each timestep of SAS execution
Adaptive Test Plans

- Network loss rate
- Data mirror capacity
- Sensor fuzz

OC 1 → SAS Config. 1

OC 2 → SAS Config. 2

ATP 2

... → ...

ATP n → OC n

SAS Config. 1 → SAS Config. 2 → SAS Config. n
Adaptive Test Plans

Comprises all test suites for given operating context

ATP 1

OC 1

SAS Config. 1

ATP 2

OC 2

SAS Config. 2

ATP n

OC n

SAS Config. n
Adaptive Test Plans

**ATP 1**

- Defined by test engineer
- Automatically derived by Proteus

**TS 1.0**

- TS1.1
- TS1.2
- TS1.3
- TS1.4

**ATP 2**

**ATP n**

**SAS Config. 1**

**SAS Config. 2**

**SAS Config. n**

**OC1**

**OC2**

... **OCn**
Test Case Activation State

- Test cases within test suite have an activation state:
  - **ACTIVE**: Executed when current test suite is performed
  - **INACTIVE**: Not executed when current test suite is performed
  - **N/A**: Not executed, as it is not relevant to current operating context

- Default test suite ($TS_{k.0}$):
  - Relevant to operating context: **ACTIVE**
  - Irrelevant test cases labeled: **N/A**
Testing cycle at each step of SAS execution

1. Execute default test suite
2. Analyze test results
3. Perform fine-grained test case parameter adaptation
4. Perform coarse-grained test suite adaptation
5. Determine if cycle is complete
   1. If complete: halt testing
   2. If not complete:
      1. Execute intermediate test suite
Test Case Fitness

- Test case fitness (relevance) is defined as:

\[
relevance_{TC} = 1.0 - \frac{|value_{measured} - value_{expected}|}{value_{expected}}
\]

- For example:

<table>
<thead>
<tr>
<th>High relevance to environment</th>
<th>Low relevance to environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case expected value = 0.50</td>
<td>Test case expected value = 0.50</td>
</tr>
<tr>
<td>Test case measured value = 0.45</td>
<td>Test case measured value = 0.01</td>
</tr>
<tr>
<td>Fitness = 0.90</td>
<td>Fitness = 0.02</td>
</tr>
</tbody>
</table>
Results Analysis

Each test case is correlated to at least one goal for validation
- Test result validated against utility value

- **True positive**
  - Test case relevance = \([\text{Threshold, 1.0}]\)
  - Utility value(s) > 0.0

- **True negative**
  - Test case relevance = \([0.0, \text{Threshold})\)
  - Utility value(s) = 0.0

- **False positive**
  - Test case relevance = \([\text{Threshold, 1.0}]\)
  - Utility value(s) = 0.0

- **False negative**
  - Test case relevance \([0.0, \text{Threshold})\)
  - Utility value(s) > 0.0

- **Error detected in test case**
  - Adapt test case

- **Error detected in both SAS and test case**
  - Adapt both

- **Error detected in SAS**
  - Perform reconfiguration

- **No action necessary**
Fine-grained Adaptation

- **Veritas** [Fredericks2014.SEAMS]
  - Adapts non-invariant false positive and false negative test cases
  - Online evolutionary algorithm
  - Searches for a better test case *expected value*
    - Addresses system or environmental uncertainty for each operating context

Data mirror capacity

| TC1 | 4.50 | TC1 | 4.65 |
Coarse-grained Adaptation

- Dynamically generate test suites based on test results
End of Testing Cycle

- Testing cycle terminates when:
  - New SAS configuration is invoked
    - New testing cycle initiated
  - All test cases result in true positives

- If the cycle continues, then the dynamically-generated test suite is executed instead of the default test suite
Case Study

- Simulated RDM network
  - [15, 30] data mirrors
  - [100, 200] data messages
  - 300 timesteps

- Uncertainty at each timestep:
  - Unpredictable network link failures
  - Randomly dropped or delayed messages
  - Noise applied to data mirror sensors / network links
Test specification:
- 36 test cases
  - 7 invariant [precluded from adaptation]
  - 29 non-invariant [can be adapted]

Compared Proteus adaptive test plans to a manually-derived Control test plan
- Control test suite:
  - All test cases from test specification relevant to each operating context
Executed false positive test cases, i.e., test case relevance = [Threshold, 1.0], utility value = 0.0

Executed false negative test cases, i.e., test case relevance = [0.0, Threshold), utility value > 0.0
Experimental Results

Executed irrelevant test cases, i.e., test case relevance = 0.0

Total number of executed test cases

Control Proteus

Cumulative Number of Executed Test Cases

0 2000 4000 6000 8000 10000

Cumulative Number of Irrelevant Test Cases

0 1 2 3 4
Adaptive testing provided by Proteus framework supported by Veritas

Test suites and test cases remain relevant to changing environmental conditions

Reduces number of irrelevant test cases executed at run time
Impact of Run-Time Testing

- Analyzed impact of our framework:
  - Execution time
  - Memory overhead
  - Requirements satisfaction
  - Number of reconfigurations
Tested **three** states:

- **(S1):** All run-time testing activities enabled
  - Proteus+Veritas enabled

- **(S2):** All run-time testing activities disabled
  - Proteus+Veritas disabled

- **(S3):** Run-time testing framework removed
  - Proteus+Veritas code / data structures removed from implementation
Execution time

- Measured total execution time of RDM simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average execution time (seconds)</td>
<td>23.030</td>
<td>13.901</td>
<td>13.785</td>
</tr>
</tbody>
</table>
Execution time

- Measured total execution time of RDM simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average execution time (seconds)</td>
<td>23.030</td>
<td>13.901</td>
<td>13.785</td>
</tr>
</tbody>
</table>

Significant (p < 0.05)
Impact

- **Execution time**
  - Measured total execution time of RDM simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average execution time (seconds)</td>
<td>23.030</td>
<td>13.901</td>
<td>13.785</td>
</tr>
</tbody>
</table>

- **Memory footprint**
  - Measured maximum memory usage of RDM simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average memory usage (megabytes)</td>
<td>65.234</td>
<td>65.332</td>
<td>65.020</td>
</tr>
</tbody>
</table>
Impact

- **Execution time**
  - Measured total execution time of RDM simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average execution time (seconds)</td>
<td>23.030</td>
<td>13.901</td>
<td>13.785</td>
</tr>
</tbody>
</table>

- **Memory footprint**
  - Measured maximum memory usage of RDM simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average memory usage (megabytes)</td>
<td>65.234</td>
<td>65.332</td>
<td>65.020</td>
</tr>
</tbody>
</table>

Not significant (p > 0.05)
Requirements satisficement
- Calculated average utility value over simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average utility value</td>
<td>0.7717</td>
<td>0.7656</td>
<td>0.7656</td>
</tr>
</tbody>
</table>
- Requirements satisficement
  - Calculated average utility value over simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average utility value</td>
<td>0.7717</td>
<td>0.7656</td>
<td>0.7656</td>
</tr>
</tbody>
</table>

Not significant ($p > 0.05$)
Impact

- Requirements satisficement
  - Calculated average utility value over simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average utility value</td>
<td>0.7717</td>
<td>0.7656</td>
<td>0.7656</td>
</tr>
</tbody>
</table>

- Number of reconfigurations
  - Averaged number of triggered RDM reconfigurations

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average number of reconfigurations</td>
<td>23.00</td>
<td>17.28</td>
<td>19.16</td>
</tr>
</tbody>
</table>
Impact

- Requirements satisficement
  - Calculated average utility value over simulation

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average utility value</td>
<td>0.7717</td>
<td>0.7656</td>
<td>0.7656</td>
</tr>
</tbody>
</table>

- Number of reconfigurations
  - Averaged number of triggered RDM reconfigurations

<table>
<thead>
<tr>
<th></th>
<th>(S1)</th>
<th>(S2)</th>
<th>(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average number of reconfigurations</td>
<td>23.00</td>
<td>17.28</td>
<td>19.16</td>
</tr>
</tbody>
</table>

Not significant (p > 0.05)
Discussion of Testing Impact

- Run-time, adaptive testing only **significantly** impacts RDM in terms of execution time
  - Exploring parallelization strategies to reduce time impact

- While not significant, a clear difference in mean utility values exist in requirements satisfaction
  - Timing of measurement causes sampling times to be slightly different
Related Work

Search-based software testing

- Techniques such as evolutionary computation, hill climbing, simulated annealing used for different testing approaches in model testing [Harman2009], regression testing [Harman2012], and structural testing [McMinn2011]
- EvoSuite [Fraser2011] and Nighthawk [Andrews2011] are evolutionary frameworks for generating test suites and instantiating unit tests
- Veritas uses a run-time evolutionary algorithm, whereas the other techniques focus on design time search

Run-time testing

- Implemented using reinforcement learning [Veanes2006], recording & replaying [Tsai1990], and Markov modeling [Filieri2011] approaches
- Veritas combines evolutionary search for test parameters with utility-based validation
- Proteus maintains relevance of test suites as conditions change
Related Work

- **Test suite generation**
  - Requirements specification used to generate formal grammars [Bauer1979]
  - Proteus generates new test suites based upon a pre-defined default test suite and executes based on monitored conditions
  - Artificial intelligence used to automatically generate test plans for graphical user interfaces [Memon2001]
  - Proteus analyzes monitoring information to select appropriate test suite

- **Test case selection and prioritization**
  - Select a representative set of test cases and prioritize their execution [Harman2009]
  - Proteus selects and executes tests at run time
  - Tropos [Nguyen2008] uses agent-based randomized testing to validate multi-agent systems
  - Proteus generates test suites targeted towards specific DAS operating contexts
NSF BEACON Center ([www.beacon-center.org](http://www.beacon-center.org))

NSF grants CCF-0820220, DBI-0939454, CNS-0854931, CNS-1305358

Ford Motor Company

General Motors
Q&A


