Model-Based Integration Testing

Michael Steindl
2010-10-25
1. Motivation
2. Testing basics
3. Common integration faults
4. State of the art integration testing
5. Model-based Integration testing
6. Conclusion
Motivation

V-Model for software development [V-Model 97]:

Requirements

Specification

Design

Module testing

Implementation

System testing

Integration testing

Evaluation of system’s compliance with requirements

Testing interaction between components

Testing components in isolation using stubs/drivers

Integration Testing (IEEE):
Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them [1].
Motivation

- **Key considerations** for the integration of software components [2]:
  
  - How to progressively combine components in subsystems
  - How to select an effective set of test cases

- **Integration testing** is a bottleneck of software development [3]:
  
  - 50% to 60% of resources are spent on testing
    - 70% of testing resources are spent on integration testing
    - 35% to 42% of total system development cost and effort is spent on integration testing

→ Importance of the integration testing process
Software is tested in order to determine if it does what it is supposed to do.

Problem: Program cannot be executed with all possible inputs from the input domain (exhaustive testing)
→ A suitable subset of test-cases must be found (in general much smaller than the input domain)

→ Key problem of software testing is to define a suitable subset of test-cases and assure its quality.

Well known strategies for defining test-cases:
• Black box testing,
• White box testing and
• Grey box testing (mixture of black and white box testing)
Testing basics: Black box testing

- **Black box testing**, also known as **functional testing** is (IEEE) [4]:

  Testing that *ignores the internal mechanism* of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

- Black box testing strategies [5]:
  - Boundary value testing
  - Equivalence partition testing
  - Random Testing
  - Decision tables-based testing
Black box testing is **based on requirements** analysis:

- requirements specification is converted into test cases
- test cases are executed (at least one test case within the scope of each requirement)

→ **Important part** of a comprehensive testing effort

But:

- Requirements documents are notoriously error-prone
- Requirements are written at a much higher level of abstraction than code
  - Much more detail in the code than the requirement

→ Test case exercises only a small fraction of the software that implements that requirement

→ Testing only at the requirements level may miss many sources of error in the software itself [6]
Testing basics: White box testing

- **White box testing**, also known as **structural testing** or **glass-box testing**, is (IEEE) [4]:

  Testing that *takes into account the internal mechanism* of a system or component.

- **Control flow testing**
  - Based on a flow graph which represents the internal structure of a program

- **Data-flow testing**
  - Based on the analysis of definition and use of variables in a program
Testing basics: White box testing

White box testing is **based on code**: 
- the entire software implementation is taken into account
- facilitates error detection even when the software specification is vague or incomplete

But:

If one or more requirement is not implemented
- White box testing may not detect the resultant errors of omission

Both white box and black box testing are important to an effective testing process [6]

Integration testing is mainly based on white box (grey box) testing
Jung et al. [8] have grouped integration faults into inconsistency classes with respect to the following aspects:

- **Syntax inconsistencies:**
  - Occur e.g. if mismatches between the data imported and the data expected exist.

- **Semantics inconsistencies:**
  - Occur if syntactically legal data may be interpreted in different ways by different components.
    - language inconsistencies (no problem in modern languages)
    - numerical inconsistencies (e.g. prefix ”Mega” means 1,000,000 or 1,048,576)
    - physical inconsistencies (e.g. metric or imperial units).
Common integration faults

- **Application-specific inconsistencies:**
  - Occur if pre-developed components may be reused such that their local functionalities do not accurately reflect the new global application context.
    - violation of state/input relations
    - data range inconsistencies

- **Pragmatic inconsistencies:**
  - Occur e.g. if global constraints of software components are violated
    - violation of time constraints
    - violation of memory constraints
Common integration faults

Integration faults

(Survey 2010: t.b.p Vogel-Verlag [17])
State of the art integration testing

Traditional approaches [5],[9]:

**Functional decomposition**-based integration testing
→ often expressed in a graph structure

- **Big-bang approach:**
  - All components are built and brought together in the system under test without regard for inter-component dependency or risk.
  - Indicated situations:
    - stabilized system, only few components added/changed since last passed test
    - small and testable system, all components passed component test
    - monolithic system which can not be exercised separately
State of the art integration testing

- Big Bang integration has a few serious disadvantages:
  - all components must be built before integration testing, faults could be detected relatively late in the development process
  - difficult debugging, no clues about fault location (every component is equally suspect)
  - not every interface fault could be detected

- Against this great odds Big Bang integration is broadly used:
  - no drivers or stubs necessary because all components are built
  - could result in quick completion of integration (under very favorable circumstances)

In its purest (and vilest) form, big bang testing is no method at all - ’Let’s fire it up and see if it works! It doesn’t of course.[10]
State of the art integration testing

- Bottom-up approach:
  - Stepwise verification
  - Components with the least number of dependencies are tested first
State of the art integration testing

Disadvantages:

- Need for drivers
- At higher stages it may difficult getting low level components to return values necessary to get complete coverage, need for stubs
- Interface faults may not be detected by testing the faulty component, but rather when the component that uses them is exercised
- Demonstration of the system is not possible till the end of the integration process

Advantages:

- No (few) stubs needed
- Simple test of error handling in case of faulty input values because they can be easy injected by drivers
- Interface problem between hardware, system software and the exercised software are detected in a early stage
State of the art integration testing

- Top-down approach:
State of the art integration testing

- **Schedule-Driven Integration**: Components are integrated according to their availability

- **Risk-Driven Integration**: Start by integrating the most critical or complex modules

- **Test-Driven Integration**: Components associated to a specific test case are identified

- **Use-Case-Driven**: Like test driven integration components associated to a specific case
Integration Strategies used in practice

(Survey 2010: t.b.p Vogel-Verlag [17])
Model-Based Integration Testing

- Refers to test case derivation from a model representing the software behavior [11]

- Testing by use of explicitly written down, sharable and reusable models for directing and even automating the selection and execution of test cases [12]

- Model-based testing has its roots on hardware testing, mainly telecommunications and avionics [13]

- “One engineer using model based tools could be as productive as ten test engineers using manual test generation [14]”

Concept of model-based testing [15]
Component Interaction Model

- Defined by Saglietti et al. in [16]
- Interaction behavior is represented by means of grey box models
- Information about behavior is extracted from UML diagrams

- Set of components $C$
- Set of internal states $S$
- Set of internal transitions $T$

  - $t = (\text{pre}(t), \text{tr}(t), \text{g}(t), \text{e}(t), \text{post}(t)) \in T$

  - $\text{pre}(t)$: denotes the state of $c$ in which $t$ is initiated
  - $\text{tr}(t)$: denotes an event (e.g. method call) initiating $t$
  - $\text{g}(t)$: denotes a predicate enabling $t$
  - $\text{e}(t)$: denotes an event (e.g. method call in $c'$) initiated by $t$
  - $\text{post}(t)$: denotes the state of $c$ at conclusion of $t$
## Component Interaction Model

### Table

<table>
<thead>
<tr>
<th>t</th>
<th>pre(t)</th>
<th>tr(t)</th>
<th>g(t)</th>
<th>e(t)</th>
<th>post(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>initial</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>init</td>
</tr>
<tr>
<td>t1</td>
<td>init</td>
<td>setCabin</td>
<td>-</td>
<td>-</td>
<td>Idle</td>
</tr>
<tr>
<td>t2</td>
<td>idle</td>
<td>cabinUp</td>
<td>isDoorClose = true</td>
<td>-</td>
<td>travelling up</td>
</tr>
<tr>
<td>t3</td>
<td>idle</td>
<td>cabinUp</td>
<td>isDoorClose = false</td>
<td>closeDoor</td>
<td>travelling up</td>
</tr>
<tr>
<td>t4</td>
<td>travelling up</td>
<td>cabinArrived</td>
<td>-</td>
<td>done</td>
<td>idle</td>
</tr>
<tr>
<td>t5</td>
<td>idle</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>final</td>
</tr>
</tbody>
</table>

### Diagrams

- **Component a**
  - **Initial State**: Initial
  - Event: `t0`
  - Transition: `t1`
  - Guard: `setCabinFloor`
  - Next State: Idle

- **Component b**
  - **Initial State**: Initial
  - Event: `t0`
  - Transition: `t1`
  - Guard: `openDoor`
  - Next State: Open

- **State Transitions**
  - `t0` to `init`
  - `t1` to `Idle`
  - `t2` to `cabinUp [isDoorClosed = true]`
  - `t3` to `cabinUp [isDoorClosed = false] / closeDoor`
  - `t4` to `t5`
  - `t5` to `final`
Component Interaction Model

- Mapping Model:
  - $e(t) = tr(t')$
  - Map = $\{(t3, t2')\}$

- Message Model
- State Model

Interface Coverage Criteria:
- Mapping criterion
  - Cover all possible mappings between interacting components
Conclusion

- All model-based testing approaches will make or break with the **availability of a complete and correct model**
- Documentation notoriously **error prone and incomplete**
- Creating models **only for deriving test-cases** cause redundant effort
- State-explosion problem → Tool support needed

- Model-based testing only **reasonable in a continuously model based development** process
- **Development of tools** guiding the complete model-based development process is one of the **most important parts** in software engineering
Thank you!
References


[17] Survey published in Elektronik Praxis