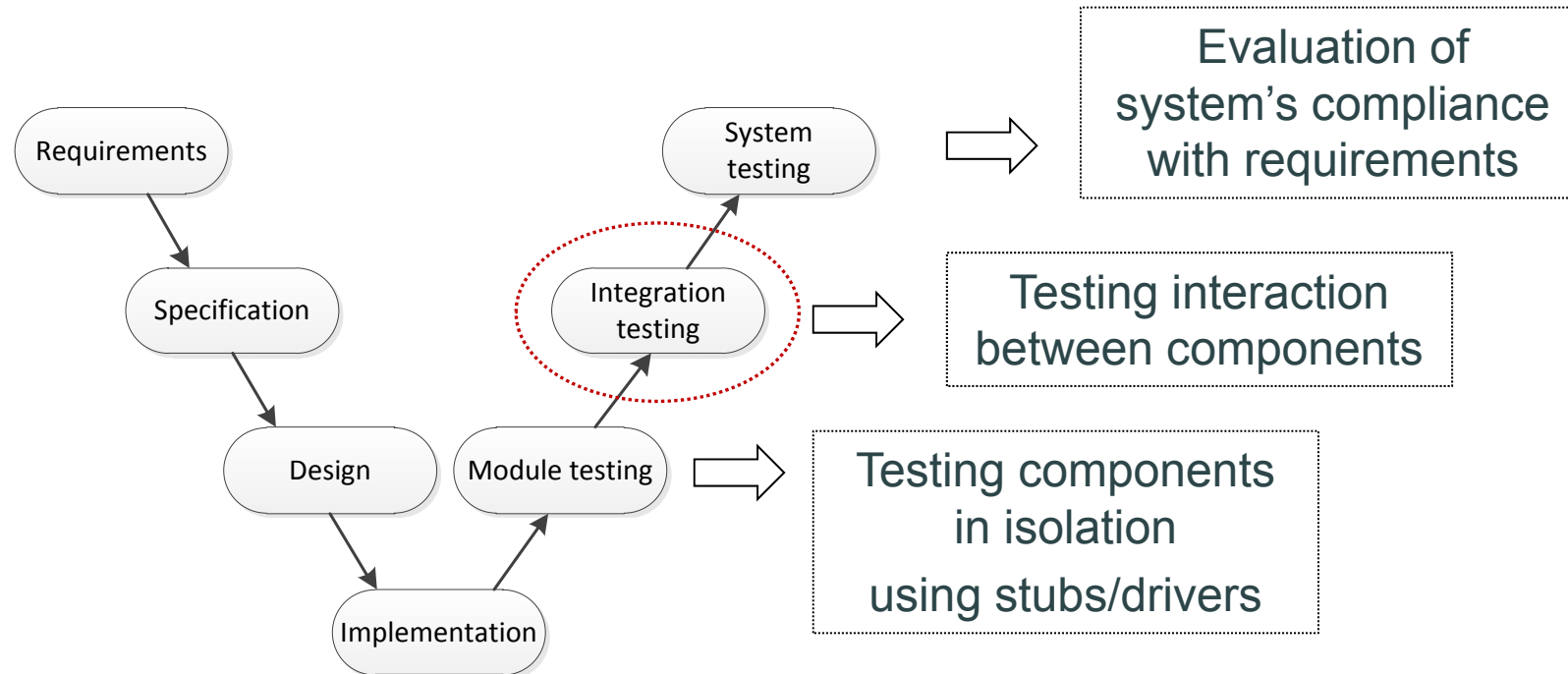


Model-Based Integration Testing

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

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



Integration Testing (IEEE):

Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them [1].

- **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)

- **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]

- **White box testing**, also know 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

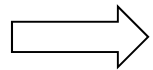
White box testing is **based on code**:

- the entire software implementation is taken into account
- facilitates error detection even when the software specification is vague or incomplete

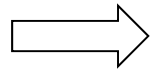
But:

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.

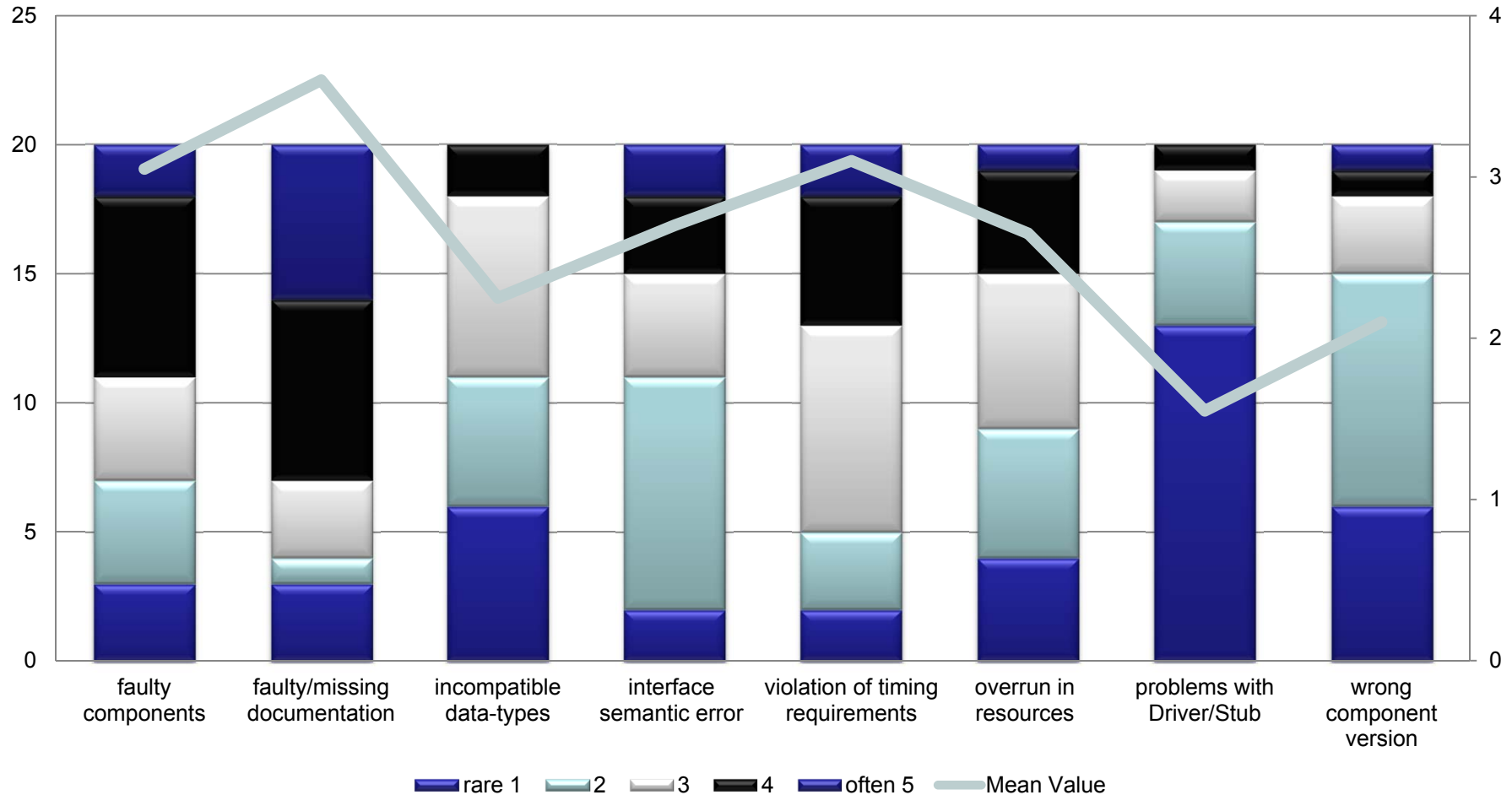
- **Semantics inconsistencies:**
 - Occur if **syntactically legal data** may be **interpreted in different** ways by different components.
 - language inconsistencies (no problem in modern languages)
 - numerical inconsistencies (e.g. prefix "Mega" means 1,000,000 or 1,048,576)
 - physical inconsistencies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

- **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])

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

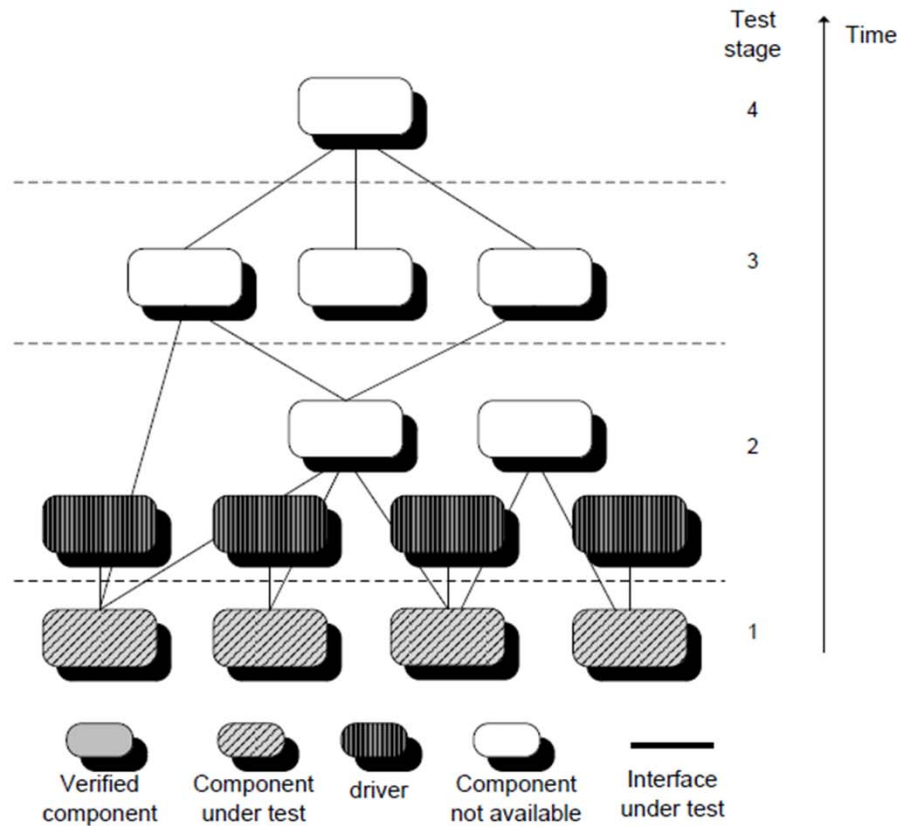
- Big Bang integration has a few serious disadvantages:
 - all components must be built before integration testing, faults could be detected relatively late in the development process
 - difficult debugging, no clues about fault location (every component is equally suspect)
 - not every interface fault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

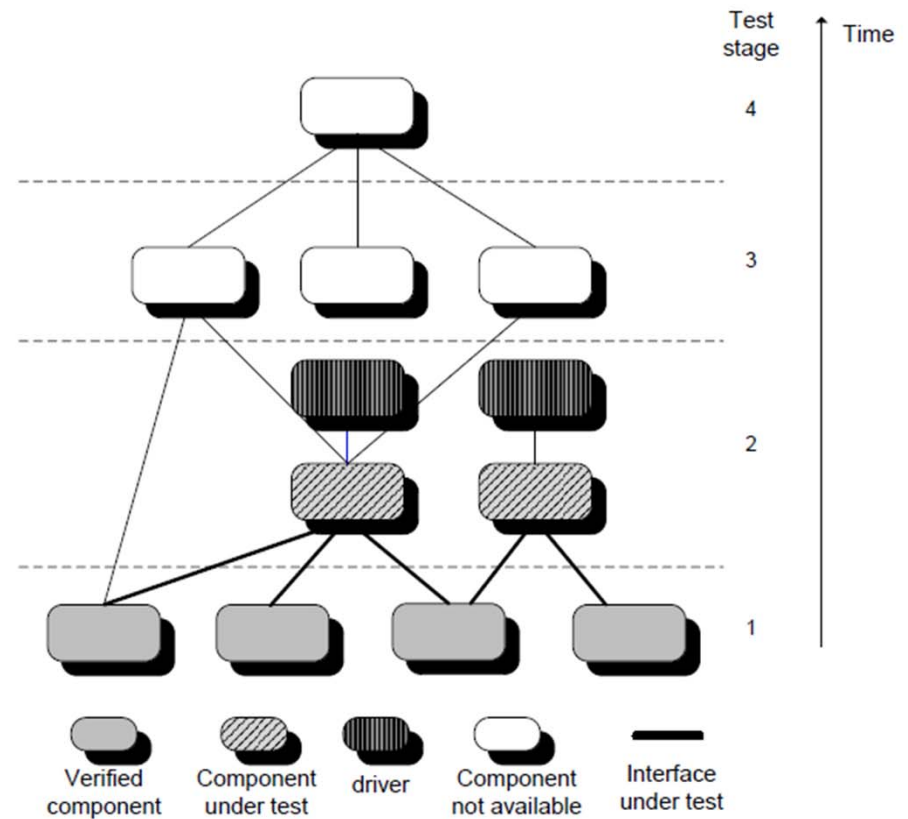
In its purest (and vilest) form, big bang testing is no method at all - 'Let's fire it up and see if it works! It doesn't of course.[10]

State of the art integration testing

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



(a) first stage



(b) second stage

Disadvantages :

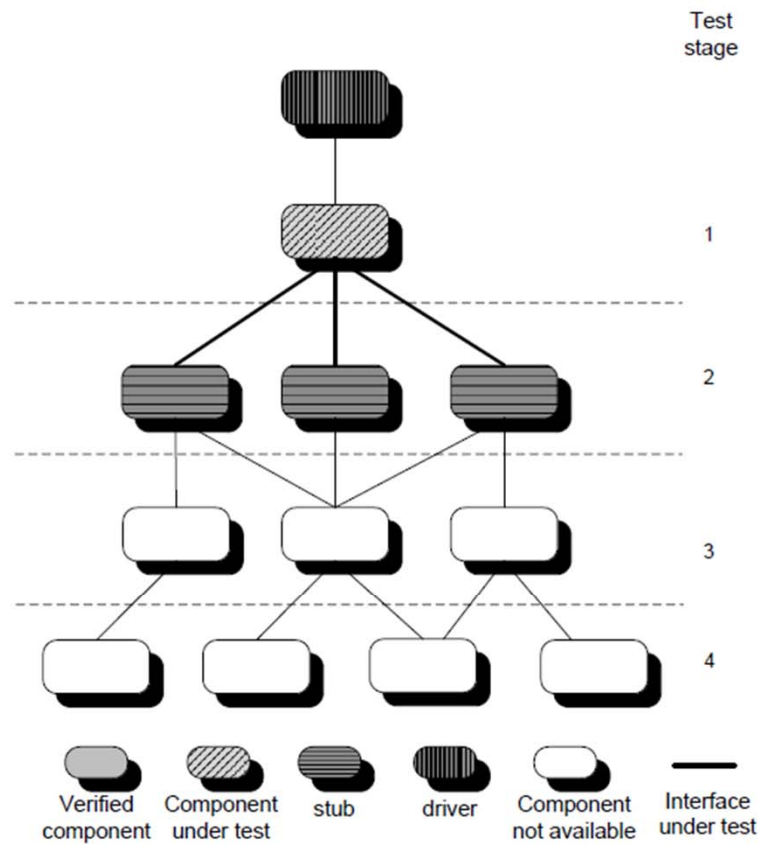
- Need for drivers
- At higher stages it may difficult getting low level components to return values necessary to get complete coverage, need for stubs
- Interface faults may not be detected by testing the faulty component, but rather when the component that uses them is exercised
- Demonstration of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

Advantages:

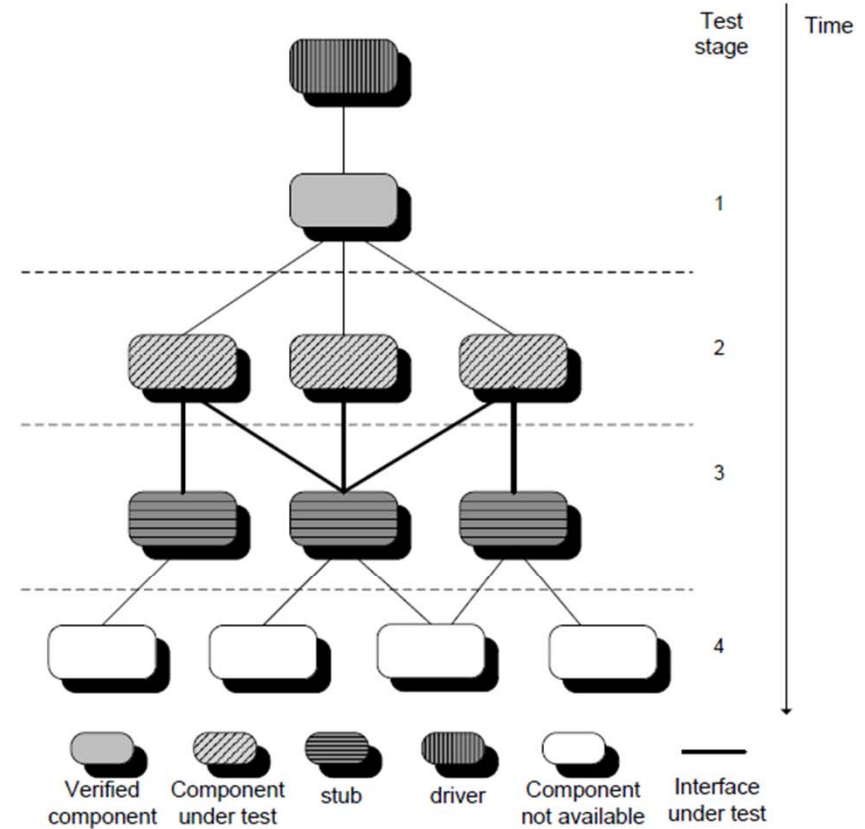
- No (few) stubs needed
- Simple test of error handling in case of faulty input values because they can be easy injected by drivers
- Interface problem between hardware, system software and the exercised software are detected in a early stage

State of the art integration testing

- Top-down approach:



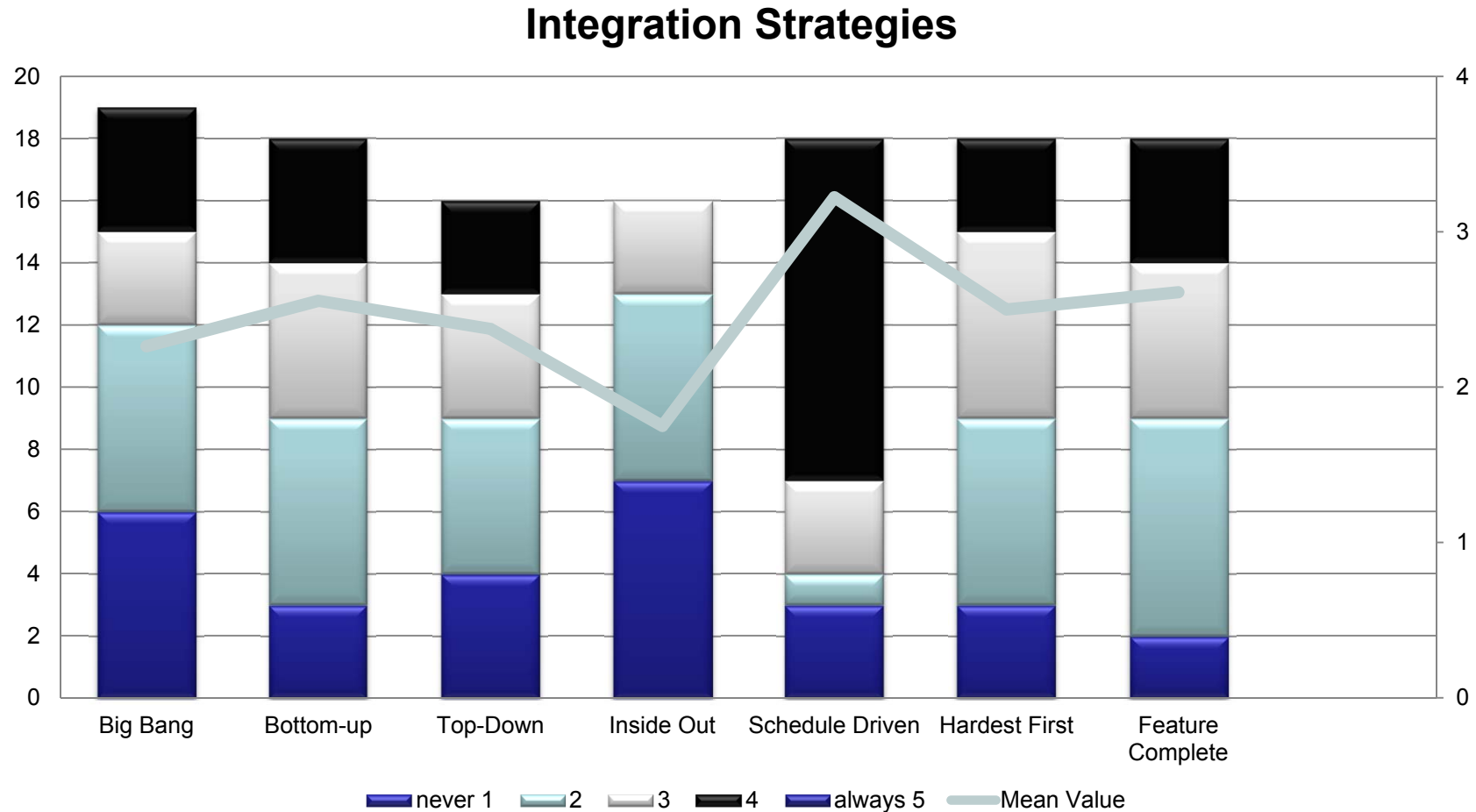
(a) first stage



(b) second stage

- **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

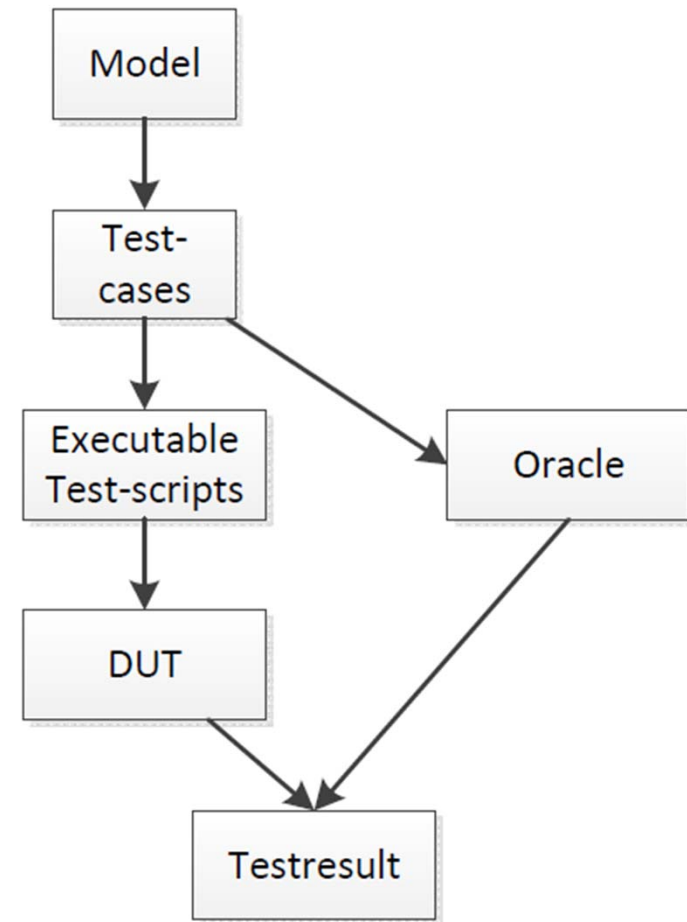
Integration Strategies used in practice



(Survey 2010: t.b.p Vogel-Verlag [17])

Model-Based Integration Testing

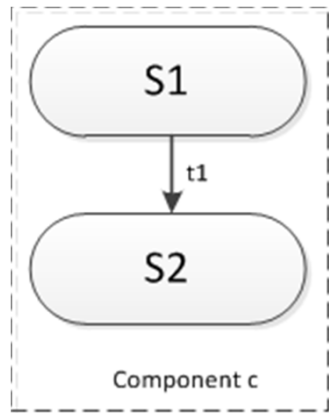
- Refers to **test case derivation from a model** representing the software behavior [11]
- Testing **by use of explicitly written down, sharable and reusable models** for directing and even automating the selection and execution of test cases [12]
- Model-based testing has its **roots on hardware testing**, 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]

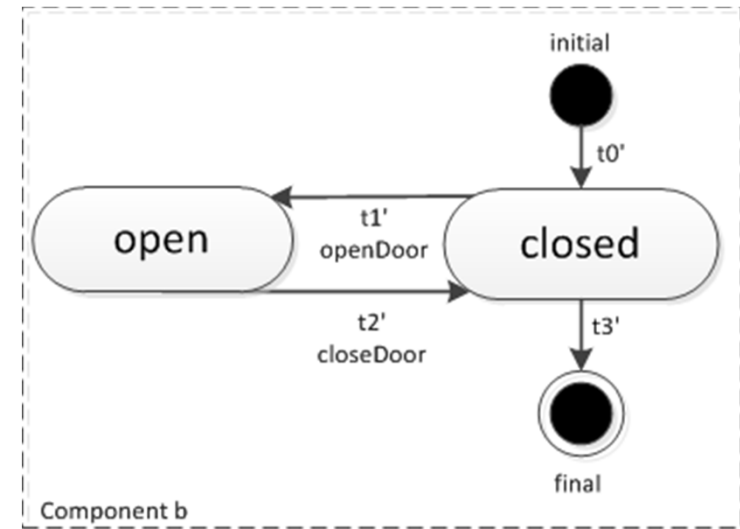
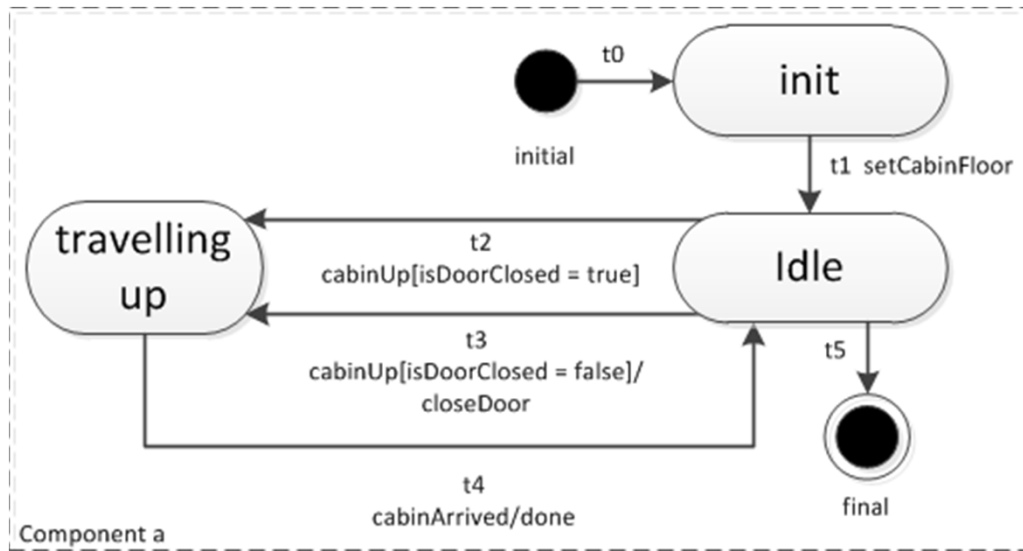
Component Interaction Model

- 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



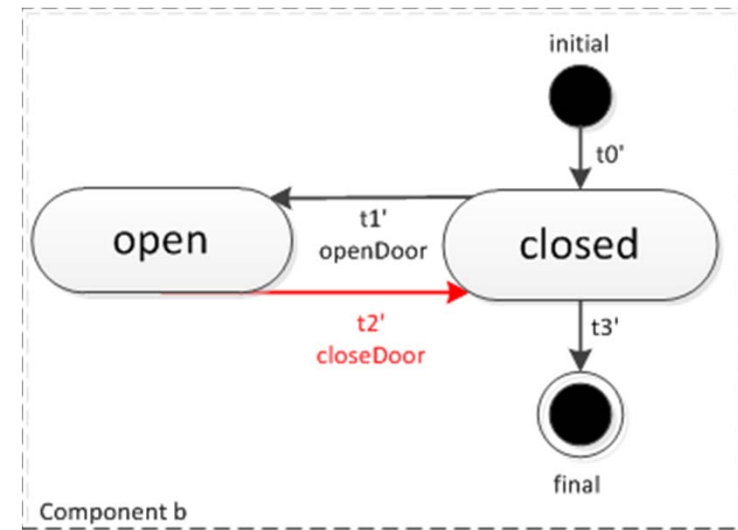
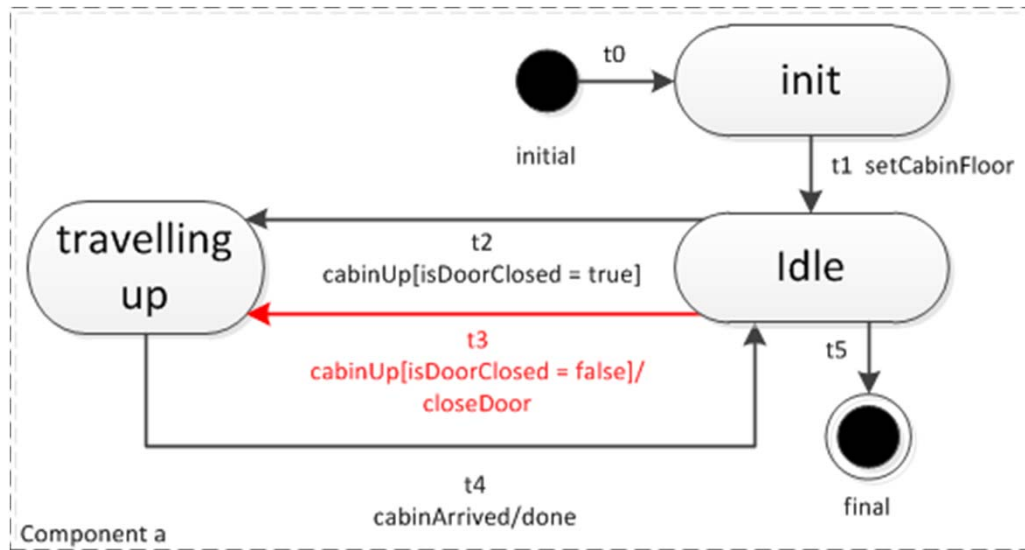
- Set of components **C**
- Set of internal states **S**
- Set of internal transitions **T**
 - $\mathbf{t} = (\mathbf{pre(t)}, \mathbf{tr(t)}, \mathbf{g(t)}, \mathbf{e(t)}, \mathbf{post(t)}) \in \mathbf{T}$
 - **pre(t)**: denotes the state of *c* in which *t* is initiated
 - **tr(t)**: denotes an event (e.g. method call) initiating *t*
 - **g(t)**: denotes a predicate enabling *t*
 - **e(t)**: denotes an event (e.g. method call in *c'*) initiated by *t*
 - **post(t)**: denotes the state of *c* at conclusion of *t*

Component Interaction Model



t	pre(t)	tr(t)	g(t)	e(t)	post(t)
t0	initial	-	-	-	init
t1	init	setCabin	-	-	Idle
t2	idle	cabinUp	isDoorClose = true	-	travelling up
t3	idle	cabinUp	isDoorClose = false	closeDoor	travelling up
t4	travelling up	cabinArrived	-	done	idle
t5	idle	-	-	-	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!

- [1] IEEE standard for software test documentation. 1998
- [2] Bertolino, A. ; Inverardi, P. ; Muccini, H. ; Rosetti, A.: An approach to integration testing based on architectural descriptions. In: iceccs Published by the IEEE Computer Society, 1997, S. 77
- [3] Paul, R.: End-to-end integration testing. In: apaqs Published by the IEEE Computer Society, 2001, S. 0211
- [4] Jay, F. ; Mayer, R.: IEEE standard glossary of software engineering terminology. In: IEEE Std 610 (1990)
- [5] Gao, J. ; Tsao, H.S.J. ; Wu, Y.: Testing and quality assurance for componentbased software. Artech House on Demand, 2003
- [6] Watson, A.H. ; McCabe, T.J. ; Wallace, D.R.: Structured testing: A testing methodology using the cyclomatic complexity metric. In: NIST Special Publication 500 (1996)
- [7] Wu, Y. ; Dai Pan, M.H.C.: Techniques for testing component-based software. In: iceccs Published by the IEEE Computer Society, 2001, S. 222
- [8] Jung, M. ; Saglietti, F.: Supporting Component and Architectural Re-usage by Detection and Tolerance of Integration Faults. (2005)
- [9] Binder, R.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Professional, 2000
- [10] Beizer, B.: Software system testing and quality assurance. 1984
- [11] Bertolino, A. ; Inverardi, P. ; Muccini, H.: Formal methods in testing software architectures. In: Formal methods for software architectures (2003), S.122–147
- [12] El-Far, I.K. ; Whittaker, J.A.: Model-based software testing. In: Encyclopedia of Software Engineering (2001)
- [13] Rosaria, S. ; Robinson, H.: Applying models in your testing process. In: Information and Software Technology 42 (2000), Nr. 12, S. 815–824
- [14] Clarke, J.M.: Automated test generation from a behavioral model. In: Proceedings of Pacific Northwest Software Quality Conference. IEEE Press Citeseer, 1998
- [15] Liggesmeyer, P.: Software-Qualität: Testen, Analysieren und Verifizieren von Software. Springer, 2009
- [16] Saglietti, F. ; Oster, N. ; Pinte, F.: Interface Coverage Criteria Supporting Model-Based Integration Testing. In: ARCS'07 (2008)
- [17] Survey published in Elektronik Praxis