Proceedings of

SPLiT 2008 – Fifth International Workshop on Software Product Line Testing

Editors: Peter Knauber, Andreas Metzger,
John McGregor

Informatik-Berichte
Hochschule Mannheim – Fakultät für Informatik

Computer Science Reports
Mannheim University of Applied Sciences – Computer Science Department

CSR 003.08
December 2008

URL: http://www.informatik.hs-mannheim.de/reports

hochschule mannheim

Organization

SPLiT is co-located with
12th International Software Product Line Conference (SPLC 2008) at Limerick, Ireland.

Workshop Chairs:

- Peter Knauber
  Mannheim University of Applied Sciences, Germany
  p.knauber@hs-mannheim.de
- Andreas Metzger
  University of Duisburg-Essen, Germany
  andreas.metzger@sse.uni-due.de
- John D. McGregor
  Clemson University, SC, USA
  johnmc@cs.clemson.edu

Program Committee:

- Luciano Baresi, Politecnico di Milano, Italy
- Hassan Gomaa, George Mason University, USA
- Georg Grütter, Robert Bosch, Germany
- Stefan Jungmayr, Teradyne Diagnostic Solutions, Germany
- Charles Krueger, BigLever Software, USA
- J. Jenny Li, Avaya Labs, USA
- Maurizio Morisio, Politecnico do Torino, Italy
- Frank Roessler, Avaya Labs, USA
- Davor Svetinovic, Lero, Ireland
- Tim Trew, NXP, The Netherlands

Table Of Contents

Introduction

Welcome
Andreas Metzger 1

Keynote

From Product[ion] Line to Wild West: Testing a Portfolio of Software Components for Diverse Customers
Tim Trew. 5

Presented Papers

Toward a Fault Model for Software Product Lines
John McGregor. 24

Validation of the Reliability Analysis Method and Tool
Anne Immonen, Antti Evesti 30

Functional Testing of Feature Model Analysis Tools. A First Step
Sergio Segura, David Benavides, Antonio Ruiz-Cort´es 36

Model-based Test Design for Software Product Lines
Erika Mir Olimpiew, Hassan Gomaa. 40

A Test-Driven Approach to Establishing And Managing Agile Product Lines
Yaser Ghanam, Shelly Park, Frank Maurer 46

Accepted Abstracts

Formalizing platformindependent Test Cases for Model-driven Software Development
Eugen Fischer, Peter Knauber 52

How SOA Testing Could Benefit from Software Product Line Testing
Andreas Metzger 53

Functional Testing of Feature Model Analysis Tools. A First Step
Sergio Segura, David Benavides, Antonio Ruiz-Cort´es 54

Model-based Test Design for Software Product Lines
Erika Mir Olimpiew, Hassan Gomaa. 55

A Test-Driven Approach to Establishing And Managing Agile Product Lines
Yaser Ghanam, Shelly Park, Frank Maurer 56

Workshop Summary from SPLC Proceedings

The 5th Software Product Lines Testing Workshop (SPLiT 2008)
Peter Knauber, Andreas Metzger, John D. McGregor 57

5th Software Product Lines Testing Workshop (SPLiT 2008)

Limerick, Ireland — September 8, 2008

at the 12th International Software Product Line Conference

"Bad News First"

Peter Knauber

Acknowledgments

Members of the Programme Committee

Sponsors

Post-Workshop

Procee-dings

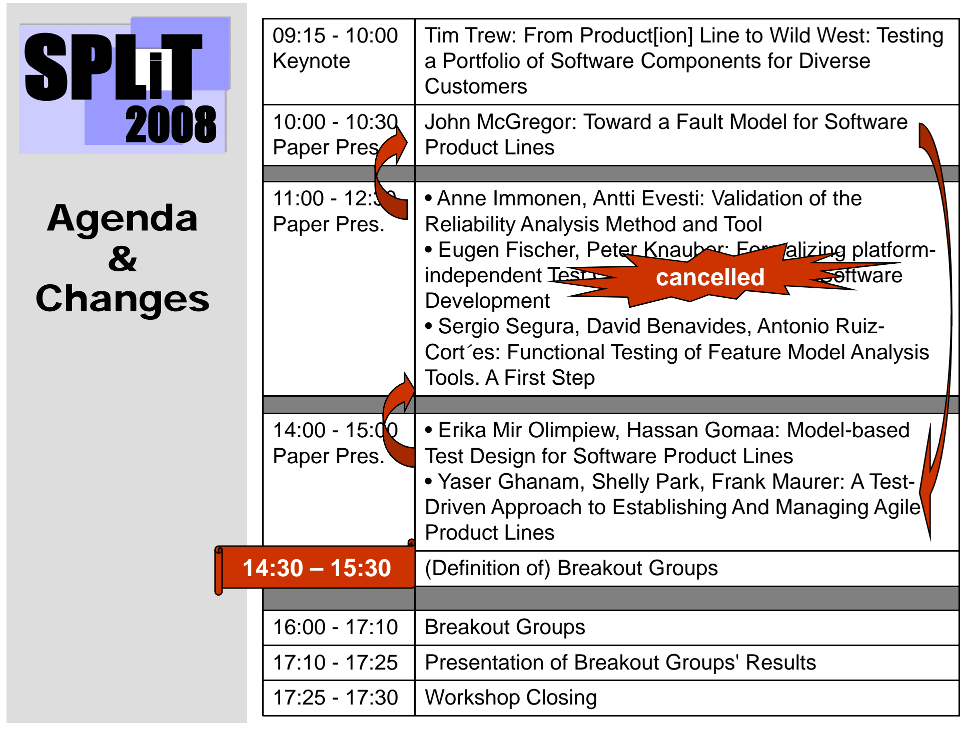
Submit your improved Versions until October, 15

"Register" now for Receiving Notification of Publication

**Intro-
duction**

Please Introduce Yourself

- **Name**
- **Affiliation**
- **Research interests**
- **Expectations from the workshop**



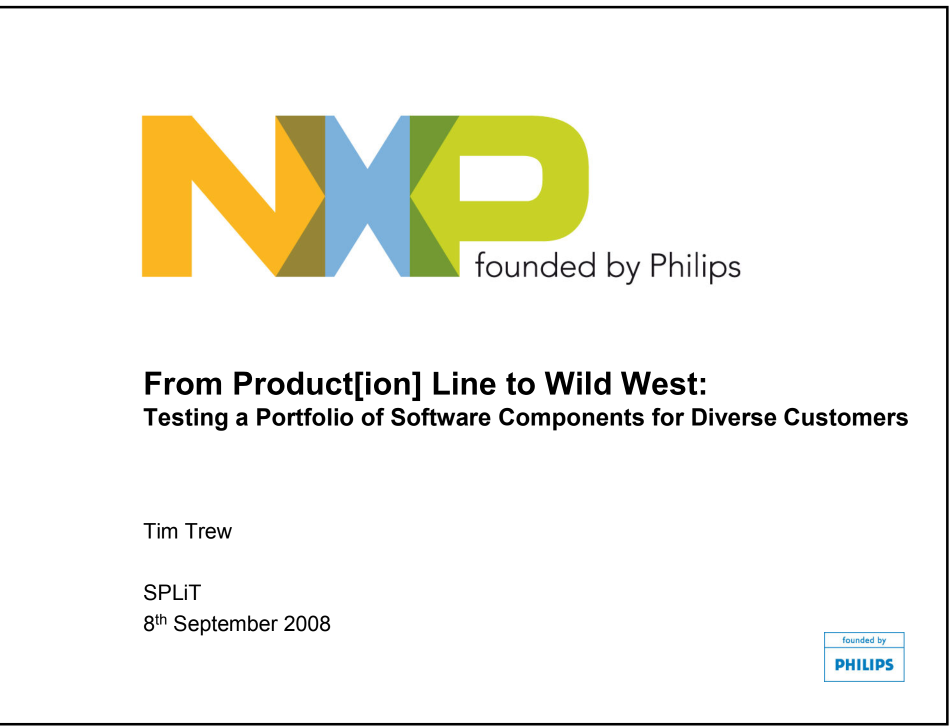
Agenda & Changes

09:15 - 10:00 Keynote	Tim Trew: From Product[ion] Line to Wild West: Testing a Portfolio of Software Components for Diverse Customers
10:00 - 10:30 Paper Pres	John McGregor: Toward a Fault Model for Software Product Lines
11:00 - 12:3[illegible] Paper Pres.	• Anne Immonen, Antti Evesti: Validation of the Reliability Analysis Method and Tool • Eugen Fischer, Peter Knauber: Formalizing platform-independent Test[illegible] Software Development • Sergio Segura, David Benavides, Antonio Ruiz-Cort´es: Functional Testing of Feature Model Analysis Tools. A First Step
14:00 - 15:00 Paper Pres.	• Erika Mir Olimpiew, Hassan Gomaa: Model-based Test Design for Software Product Lines • Yaser Ghanam, Shelly Park, Frank Maurer: A Test-Driven Approach to Establishing And Managing Agile Product Lines
14:30 – 15:30	(Definition of) Breakout Groups
16:00 - 17:10	Breakout Groups
17:10 - 17:25	Presentation of Breakout Groups' Results
17:25 - 17:30	Workshop Closing

5th Software Product Lines Testing Workshop (SPLiT 2008)

Limerick, Ireland — September 8, 2008

at the 12th International Software Product Line Conference

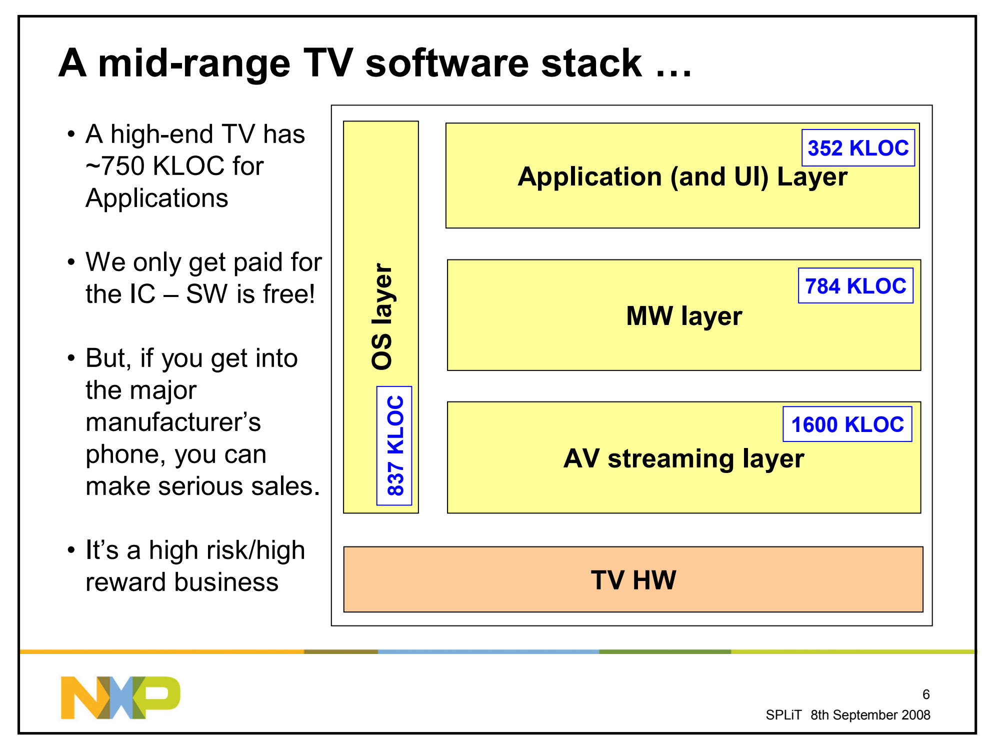

founded by Philips
From Product[ion] Line to Wild West:
Testing a Portfolio of Software Components for Diverse Customers
Tim Trew
SPLiT
8th September 2008
founded by
PHILIPS

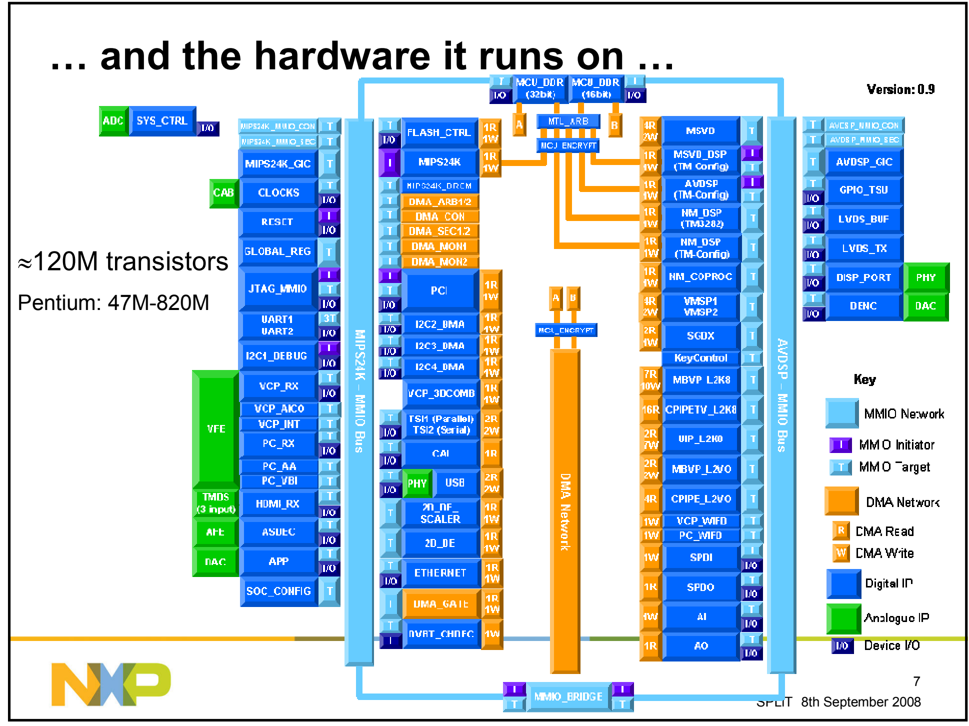
PHILIPS
TV – Dimensions of diversity
Software Product Lines
Product Line Hall of Fame
Philips Product Line of Software for Television Sets
Philips is one of the world's largest consumer electronics companies, and a global leader in televisions and other consumer products. While initially solely consisting of hardware, TVs now contain fully equipped embedded computers to control the hardware and to implement extra features.
PHILIPS
Sound
Region
Data Processing
User
WILD WEST
COLORING BOOK

Outline

- Our market and the role of software
- What's the difference in viewpoint between:
  - A product integrator
  - An [integrated circuit] component supplier
- Relationship between development and test effort
  - How much software should we deliver?
  - Do Service-Based Architectures (=Embedded SOA) help?
- Achieving code coverage in product line development

Markets for Chips+Software
5
SPLiT 8th September 2008


A mid-range TV software stack …
• A high-end TV has ~750 KLOC for Applications
• We only get paid for the IC – SW is free!
• But, if you get into the major manufacturer's phone, you can make serious sales.
• It's a high risk/high reward business
OS layer
837 KLOC
Application (and UI) Layer
352 KLOC
MW layer
784 KLOC
AV streaming layer
1600 KLOC
TV HW
6
SPLiT 8th September 2008

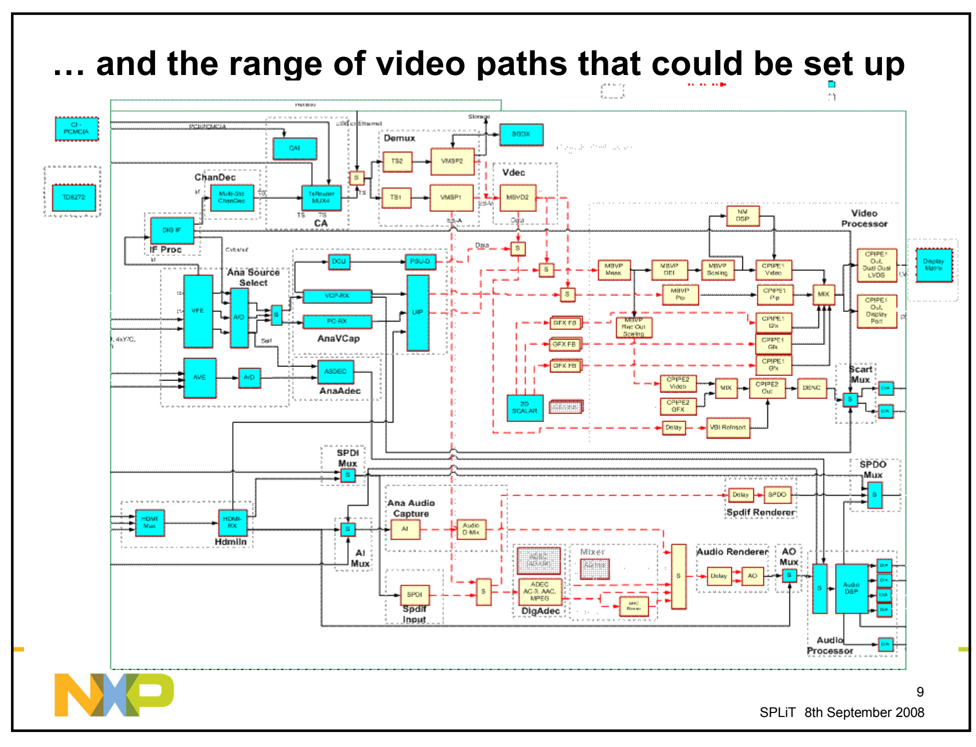


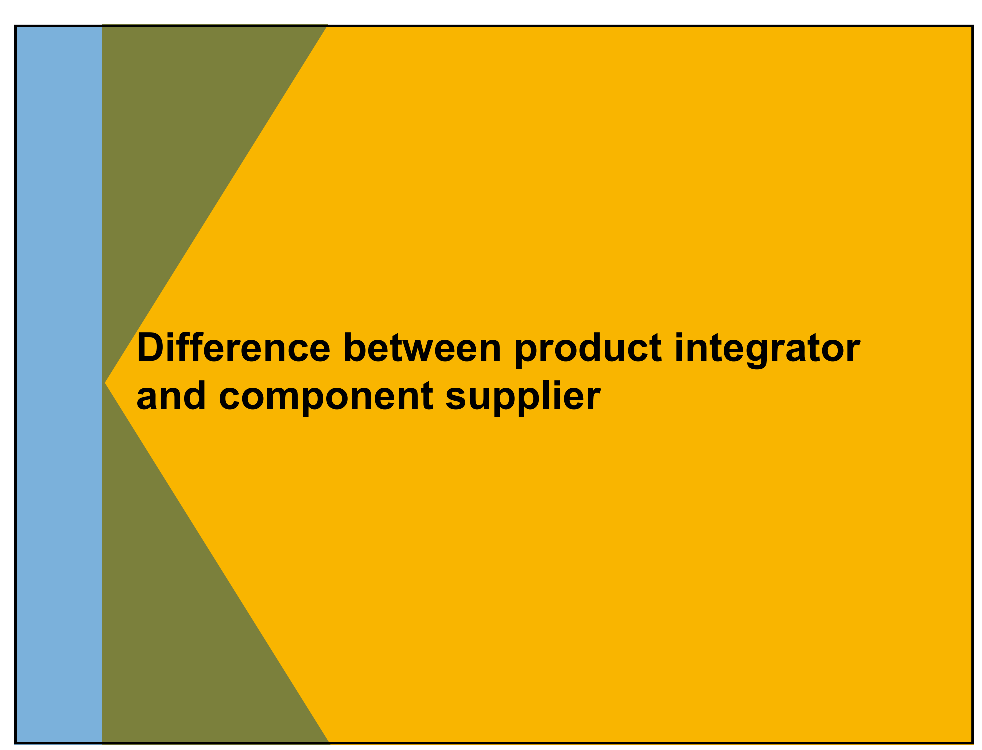
… and the hardware it runs on …

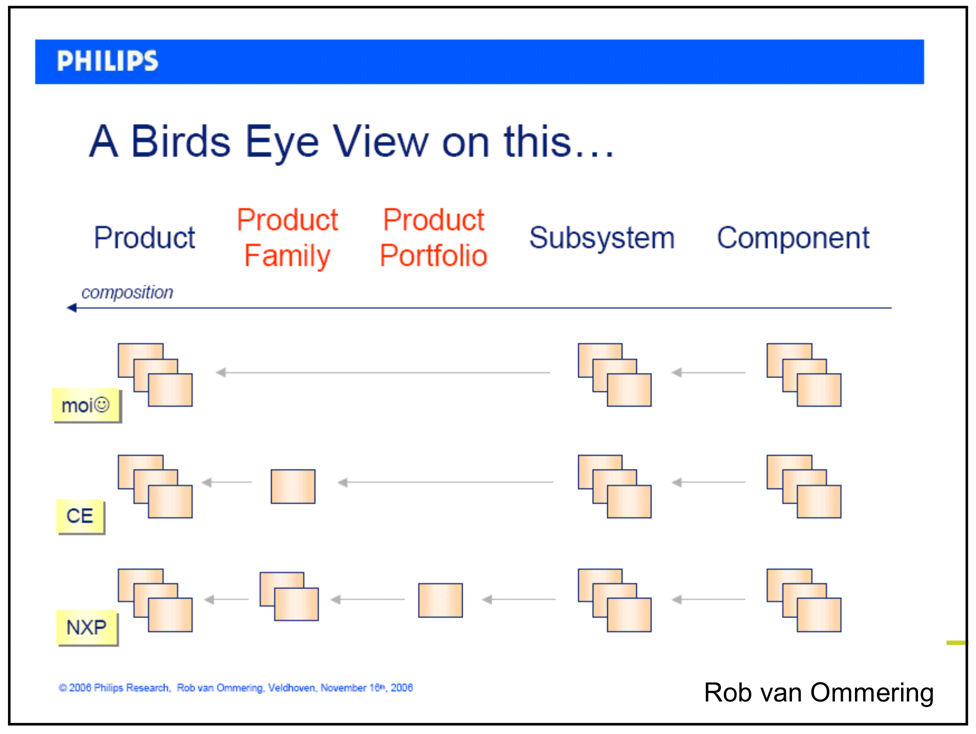
- 2 MIPS, 3 TriMedia VLIWs, 100 DMA channels, 2 memory busses + …
- Performance is critical
  - = CPU cycles + cache misses + (SDRAM + bus) latency
- Power is becoming important
  - You don't want a fan in a TV
- It's critical in a phone
  - Power down everything you don't need right now
  - Use voltage/frequency scaling
    - make the clock frequencies just high enough to meet the deadlines (dynamically)
    - Make the supply voltage just high enough to operate at that supply frequency
    - $P \propto fV^2$
- The testers have to ensure that, with all this, the software gives the right answer at the right time (**always!**)

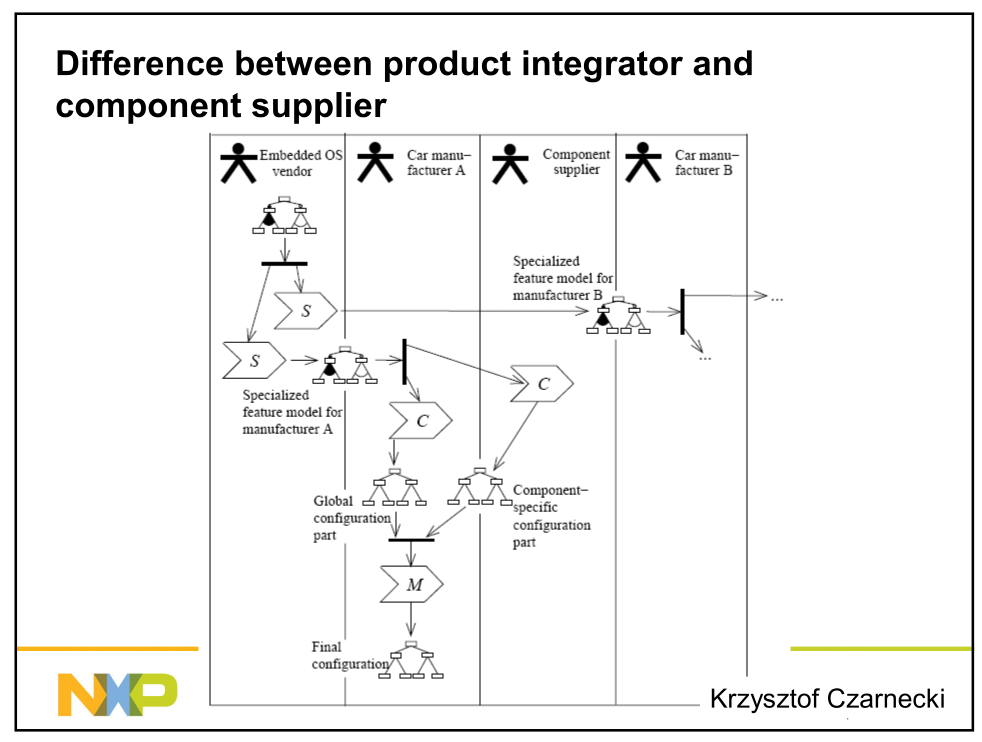
8

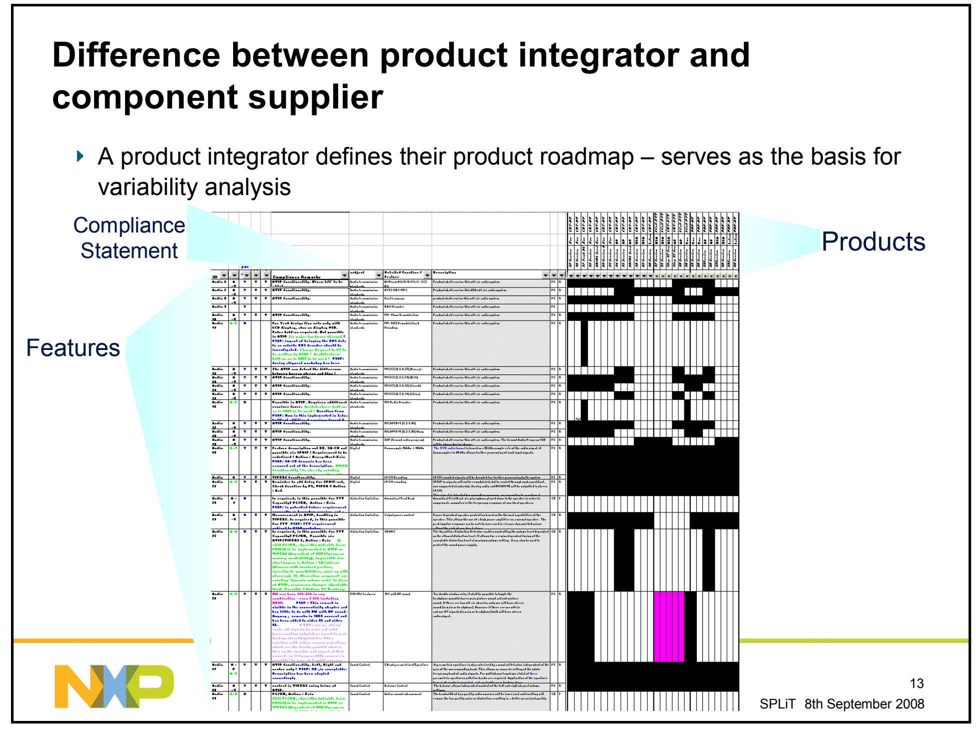
SPLiT 8th September 2008

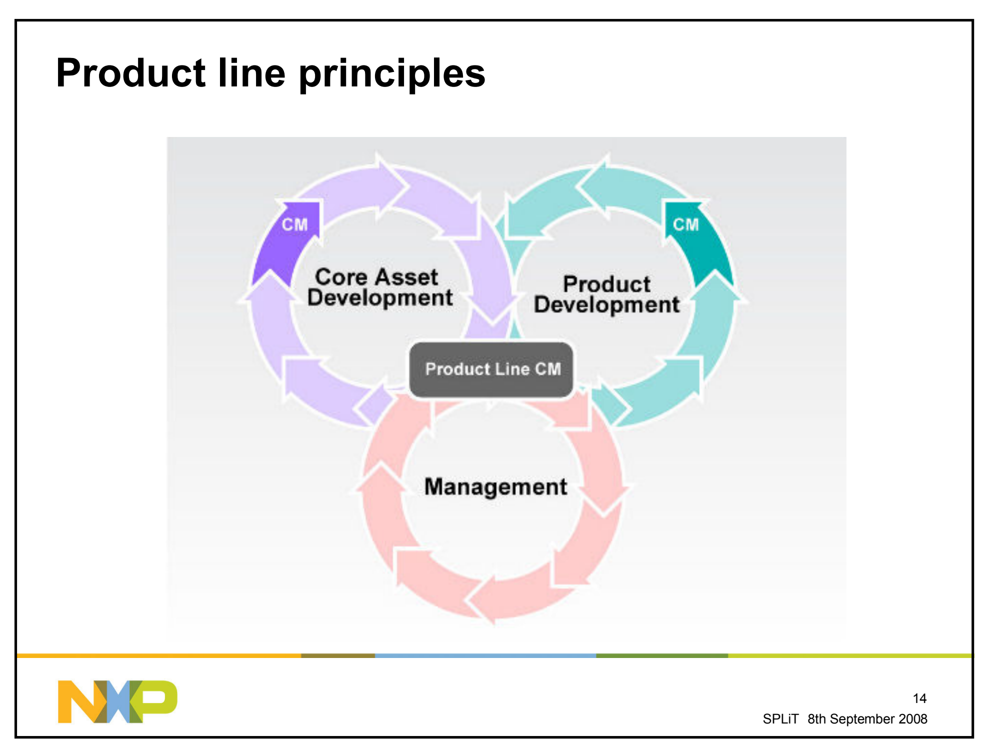

… and the range of video paths that could be set up
Demux
Vdec
ChanDec
CA
IF Proc
Ana Source Select
AnaVCap
AnaAdec
Video Processor
Scart Mux
SPDI Mux
SPDO Mux
Ana Audio Capture
Spdif Renderer
HdmiIn
AI Mux
Audio Renderer
AO Mux
Spdif Input
DigAdec
Audio Processor
9
SPLiT 8th September 2008

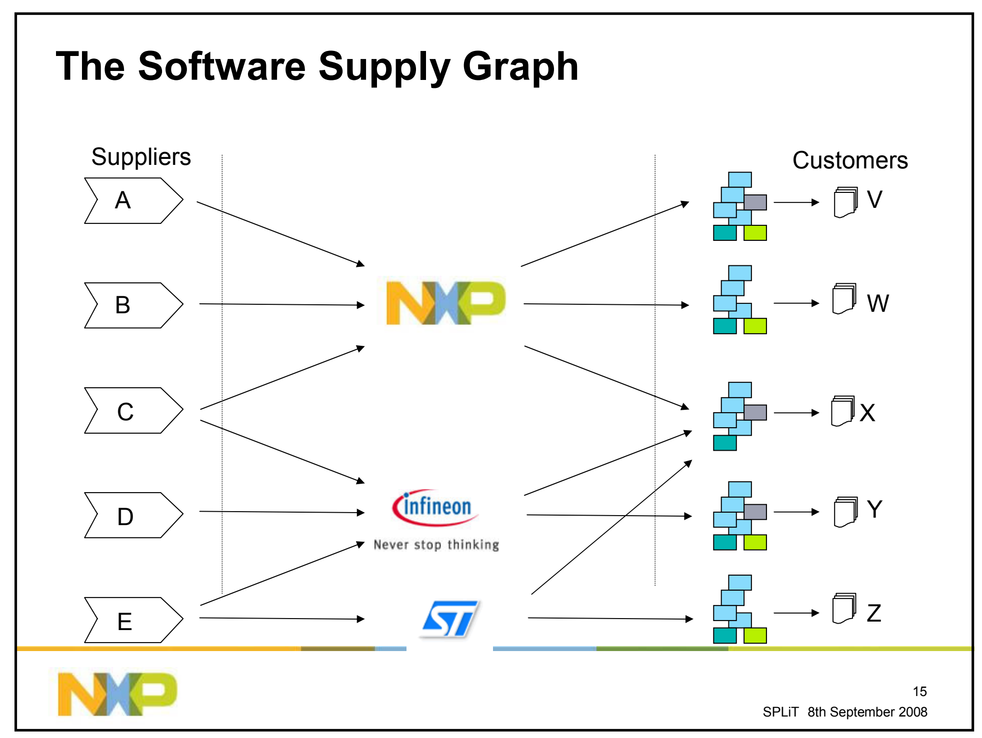

Difference between product integrator and component supplier

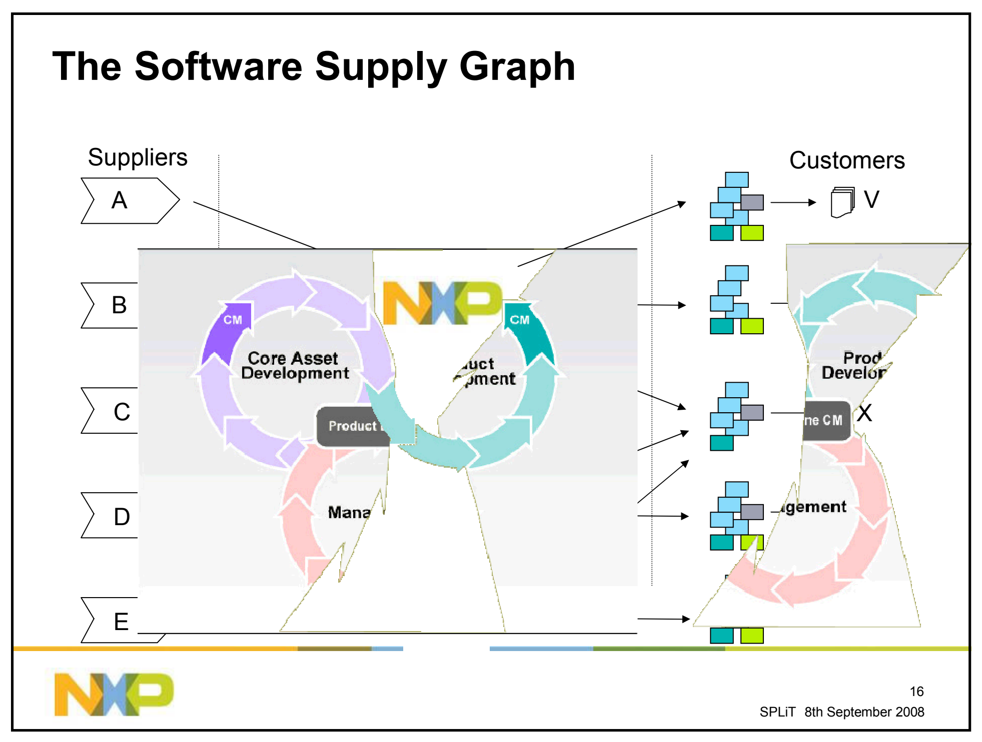

PHILIPS
A Birds Eye View on this…
Product
Product Family
Product Portfolio
Subsystem
Component
composition
moi☺
CE
NXP
© 2006 Philips Research, Rob van Ommering, Veldhoven, November 16th, 2006
Rob van Ommering

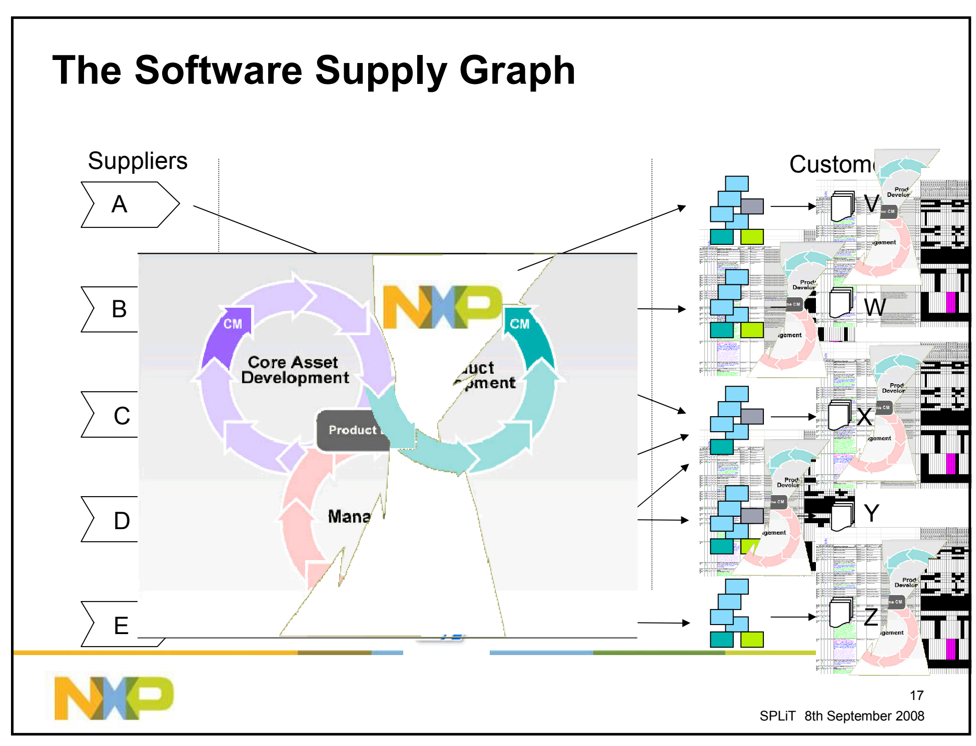

Difference between product integrator and component supplier
Embedded OS vendor
Car manu-facturer A
Component supplier
Car manu-facturer B
Specialized feature model for manufacturer B
Specialized feature model for manufacturer A
S
C
M
Global configuration part
Component-specific configuration part
Final configuration
Krzysztof Czarnecki

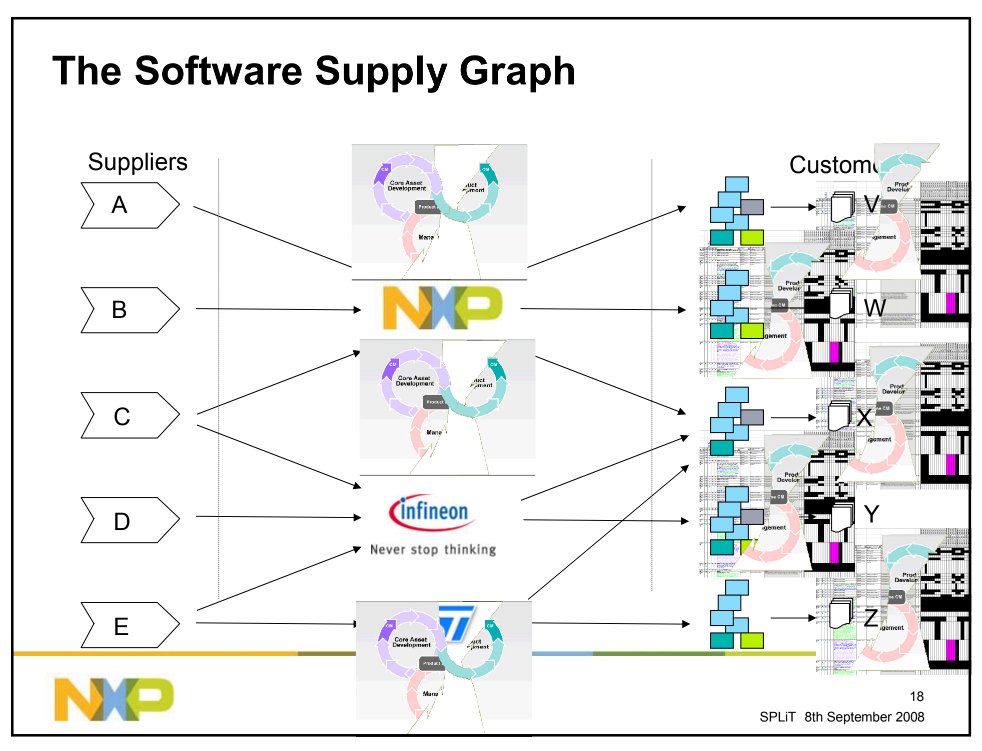

Difference between product integrator and component supplier
A product integrator defines their product roadmap – serves as the basis for variability analysis
Compliance Statement
Products
Features
13
SPLiT 8th September 2008

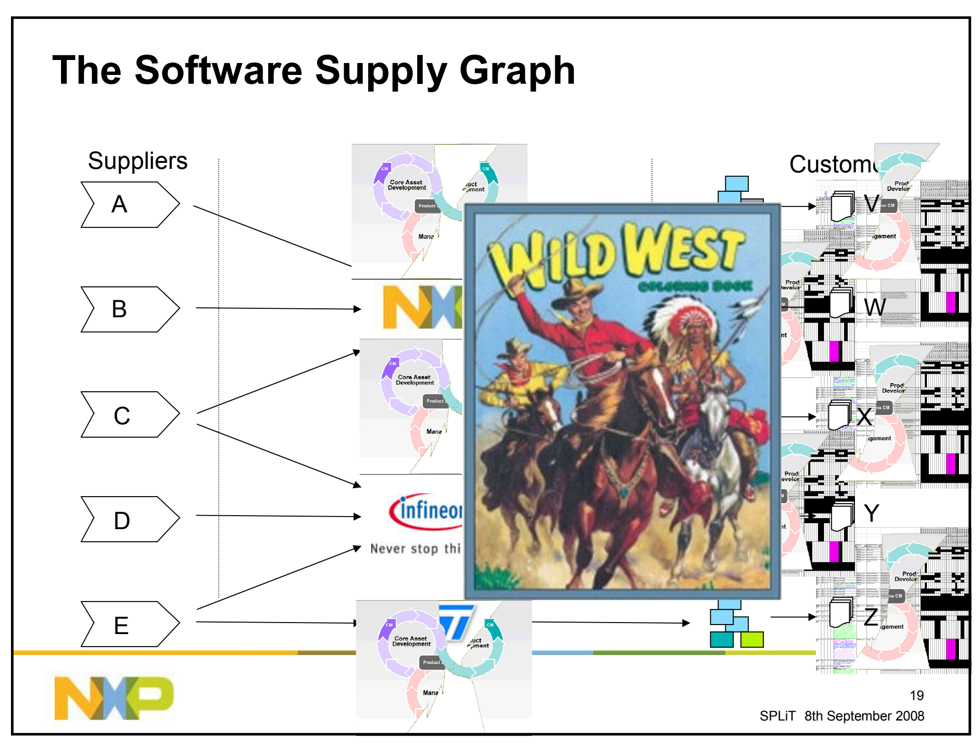

Product line principles
CM
Core Asset Development
CM
Product Development
Product Line CM
Management
14
SPLiT 8th September 2008


The Software Supply Graph
Suppliers
A
B
C
D
E
Customers
V
W
X
Y
Z
Never stop thinking
15
SPLiT 8th September 2008

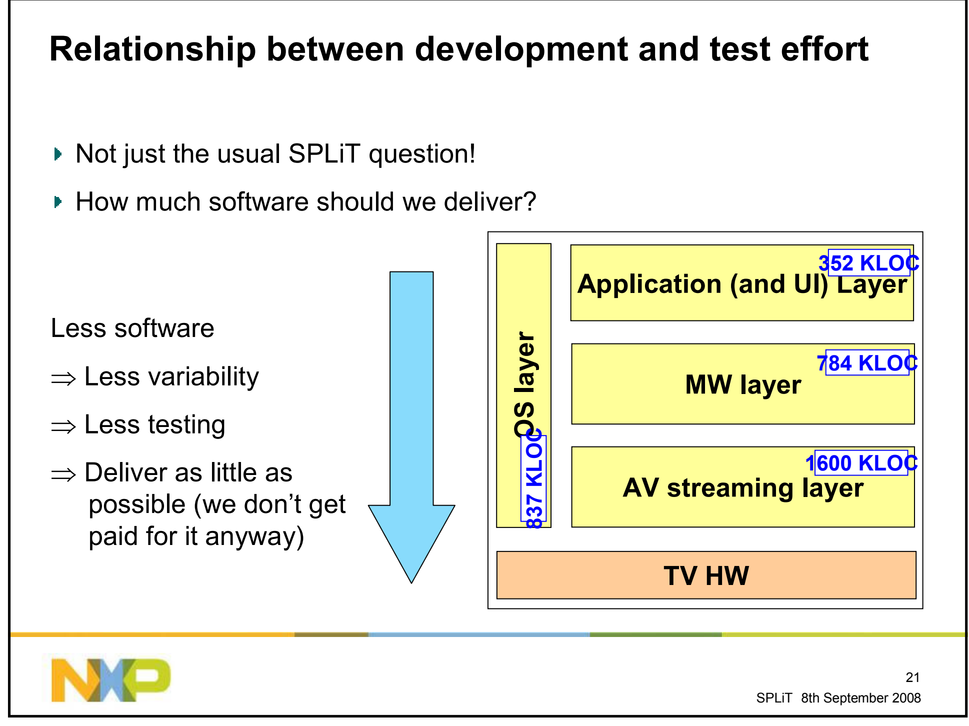

The Software Supply Graph
Suppliers
A
B
C
D
E
Customers
V
X
CM
Core Asset
Development
Product
CM
16
SPLiT 8th September 2008

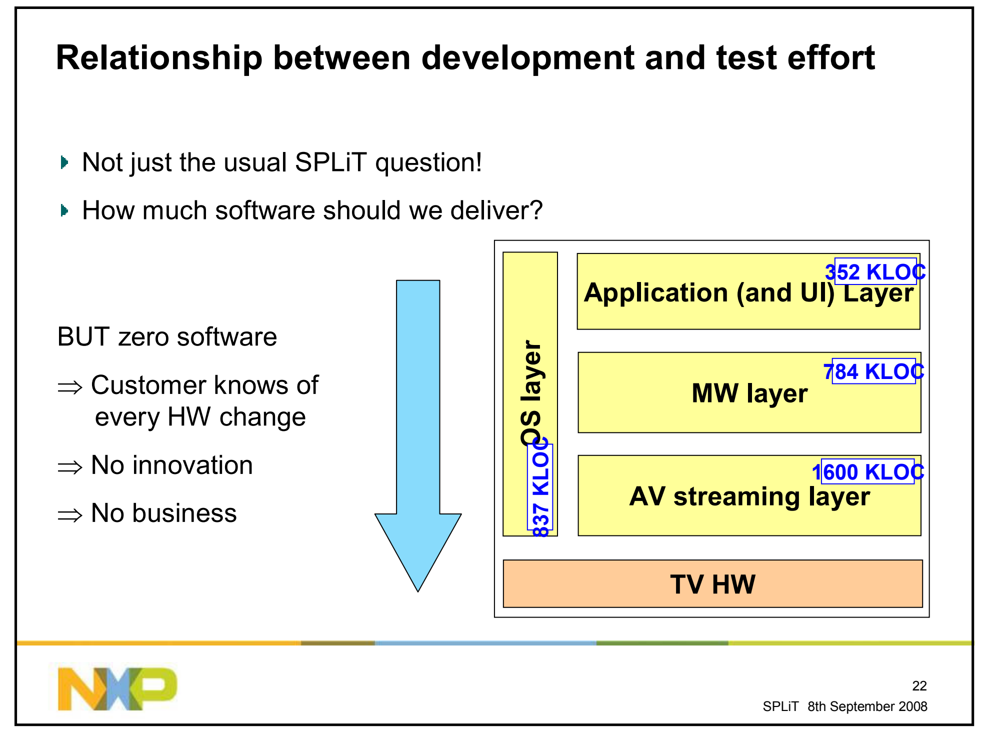

The Software Supply Graph
Suppliers
A
B
C
D
E
Custom
V
W
X
Y
Z
Core Asset Development
CM
Product
Mana
17
SPLiT 8th September 2008

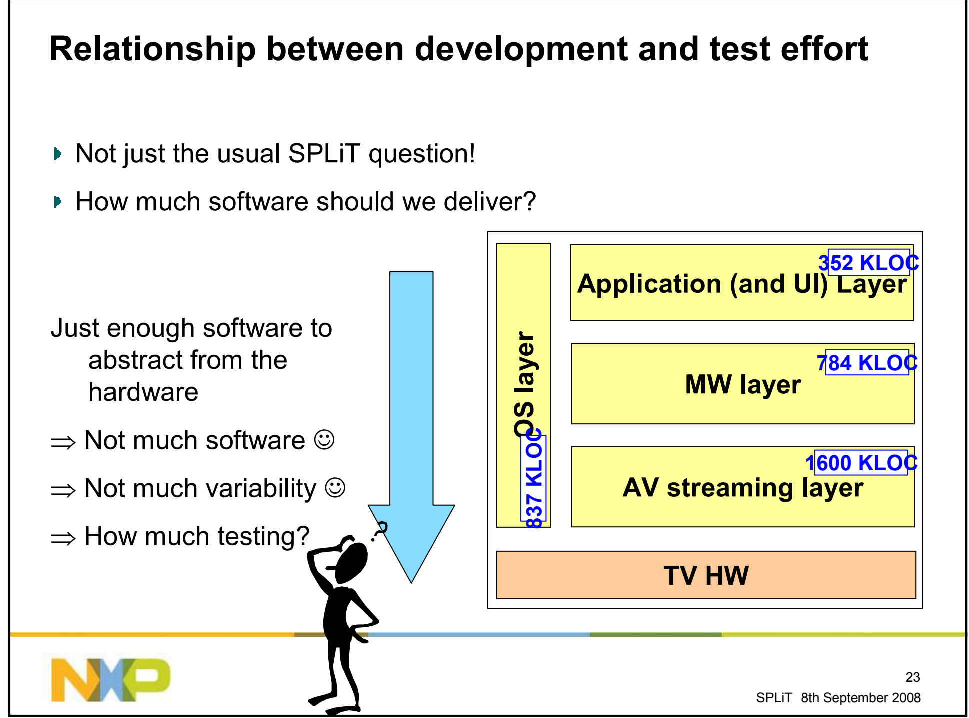

The Software Supply Graph
Suppliers
A
B
C
D
E
Custom
V
W
X
Y
Z
infineon
Never stop thinking
18
SPLiT 8th September 2008


The Software Supply Graph
Suppliers
A
B
C
D
E
Custom
V
W
X
Y
Z
19
SPLiT 8th September 2008

Relationship between development and test effort
When developing only part of the software

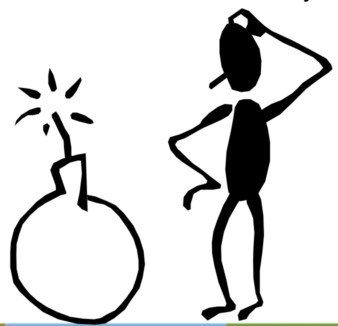

Relationship between development and test effort
Not just the usual SPLiT question!
How much software should we deliver?
Less software
⇒ Less variability
⇒ Less testing
⇒ Deliver as little as possible (we don't get paid for it anyway)
OS layer
837 KLOC
Application (and UI) Layer
352 KLOC
MW layer
784 KLOC
AV streaming layer
1600 KLOC
TV HW
21
SPLiT 8th September 2008


Relationship between development and test effort
Not just the usual SPLiT question!
How much software should we deliver?
BUT zero software
⇒ Customer knows of every HW change
⇒ No innovation
⇒ No business
OS layer
837 KLOC
Application (and UI) Layer
352 KLOC
MW layer
784 KLOC
AV streaming layer
1600 KLOC
TV HW
22
SPLiT 8th September 2008



Relationship between development and test effort

- How do we know that the user won't set up a signal path that was never tested
  - The performance requirements are quite different depending on the type (e.g. originating from film or TV camera) and resolution of the video
- How do we know that our software will work for any order in which the client gives commands?
  - Ummm …



But SOA/REST solves the world's problems!

- Commands → Requests
  Chatty → Batch
- Client requests final configuration – we decide how to get there
  - We know all the configurations that can be requested ☺
  - We only receive one message – no ordering problems ☺
  - We only have to run (# black box states) × (# white box states) test cases ☺
- BUT
  - This interface is at a **much** higher level
    ⇒ **much** more software
  - Customers care **how** we make state transitions, not just the final state
    ⇒ lots of variation points (≈ WS-Policy)
    ⇒ lots of retesting/customer (unless SPLiT knows better!)
- And all this (unpaid!) development to save some application testing
- Life was much easier when it was just Philips and we **had** to develop **all** the code!

25
SPLiT 8th September 2008

Code coverage in product lines (a problem for all of us)

- If QA set an *n*% [statement|branch|condition|MCDC|LCSAJ] coverage target
  - What does it mean for a product line?
  - How do we know if we have achieved it?
  - What's the most appropriate test strategy (testing during domain engineering *vs.* application engineering)?
- Any single application may contain only a small proportion of the code
  - Can't get a high coverage figure of *all the code base* by testing one application
  - E.g. OSAL for [MIPS|ARM|TriMedia|x86] running [VxWorks|Linux|pSOS] with [xx|yy|zz] host-target debug mechanisms, different numbers of clocks, [PCI|USB|…], [SDEaa|SDEbb|SDEcc], [basic MMU|full MMU], …

Code coverage in product lines

- Do we say that the coverage should apply to the code that is active in each application?
- Feasible for compile-time variation points
  - Example of Koala optional interfaces

```
#if( vps_iPresent_CONSTANT )
        if (fieldVPS != vps_InvalidRequestId)
        {
            vps_UnRequestDataFields ( fieldVPS );
            fieldVPS = vps_InvalidRequestId;
        }
#else
         fieldVPS = vps_InvalidRequestId;
#endif
```

- But we would have to test everything during application engineering
- And re-test for each configuration …

Code coverage in product lines

- But what if we have optional interfaces with run-time binding?
  COM example:

```
CoCreateInstance(rclsid, NULL, dwClsContext, riid, ppvObj);
result = ppvObj->QueryInterface(optInterfaceID, ppvOptObj);
if (result != E_NOINTERFACE) {
   result = ppvOptObj.m1( … )
   if (result != S_OK) {
      … // Do some error handling
   }
} else {
   …  // Take some default action
}
```

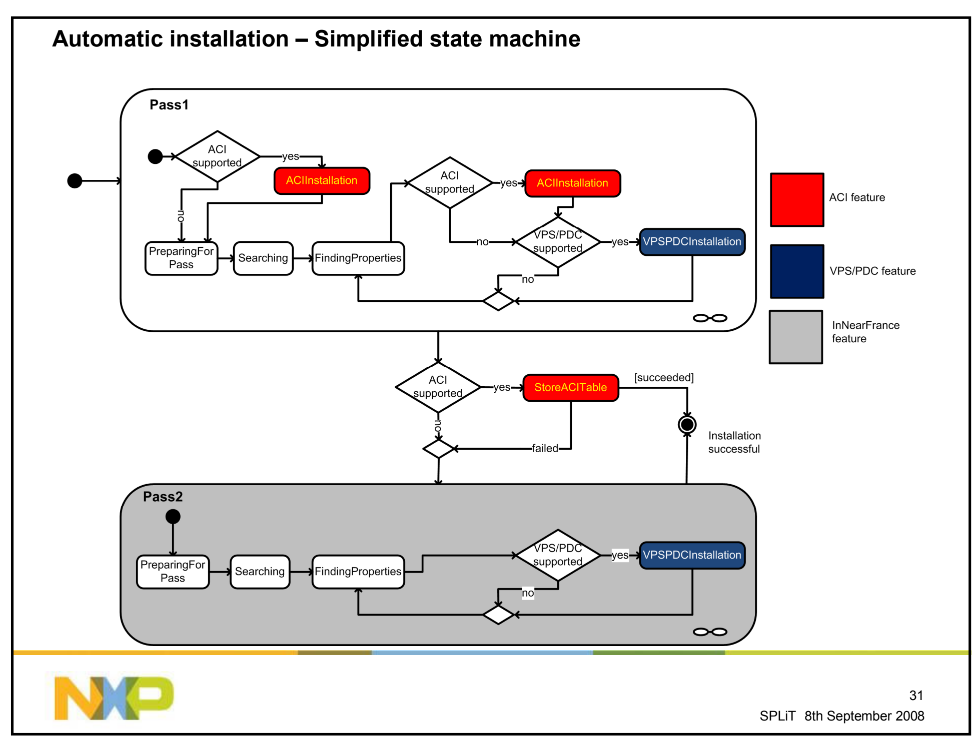
Unreachable in this application

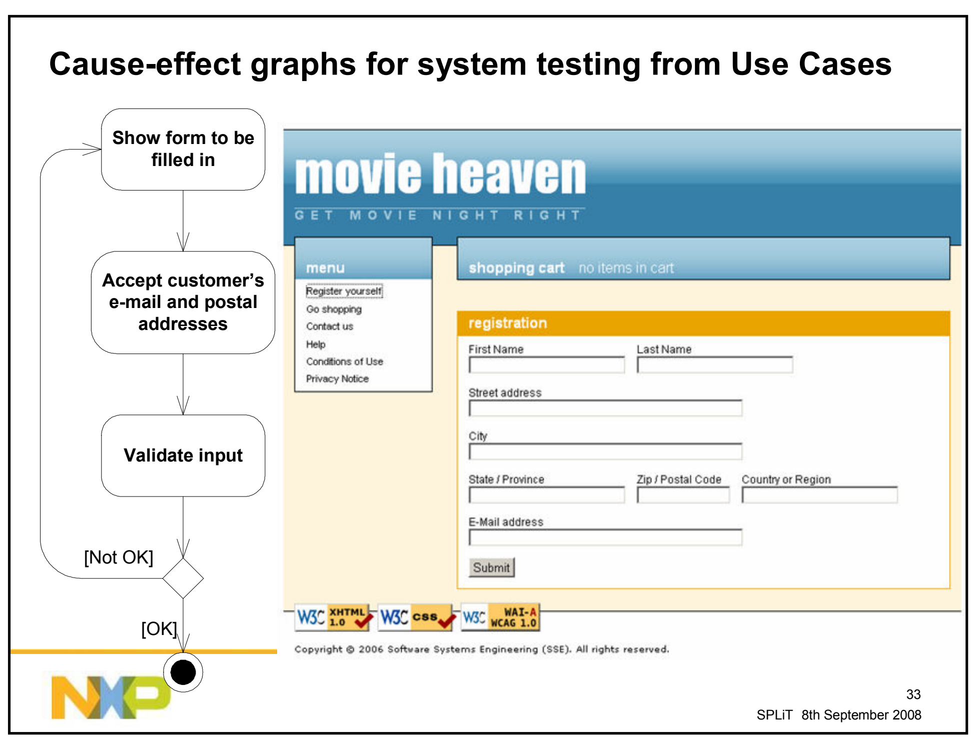
- When my coverage tool reports % coverage during application testing
  - How do I know that these statements shouldn't be included among 3.5MLOC?
    - How do I get a mapping from feature model → available interfaces with run-time binding?
  - How do I tell the tool that this decision, and all decisions in the conditional block, should be ignored

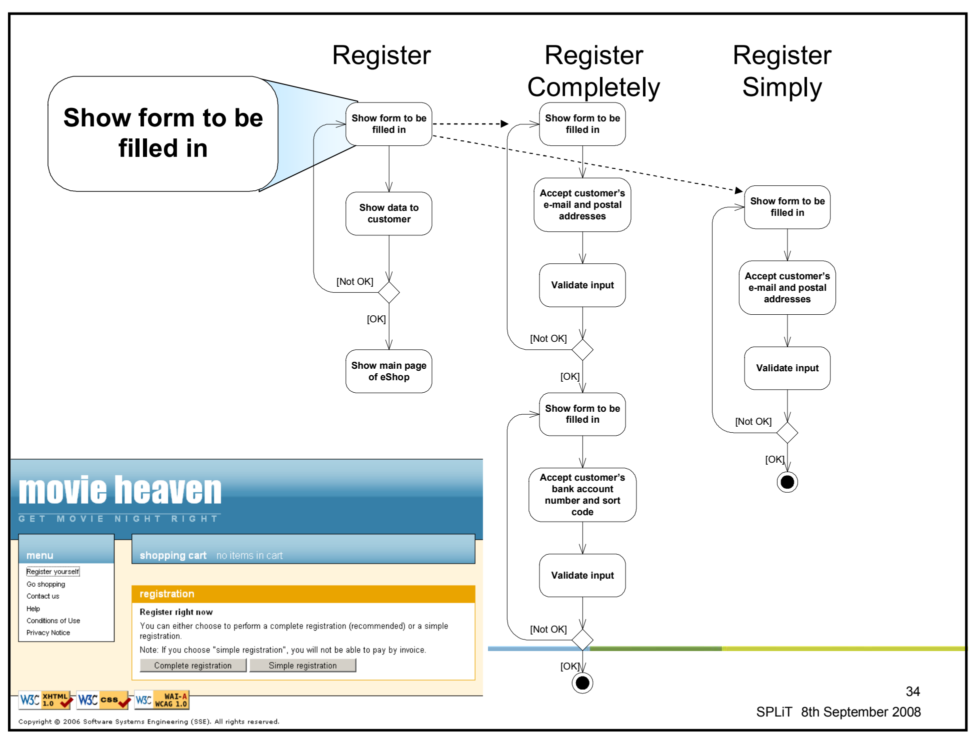
Testing during domain engineering

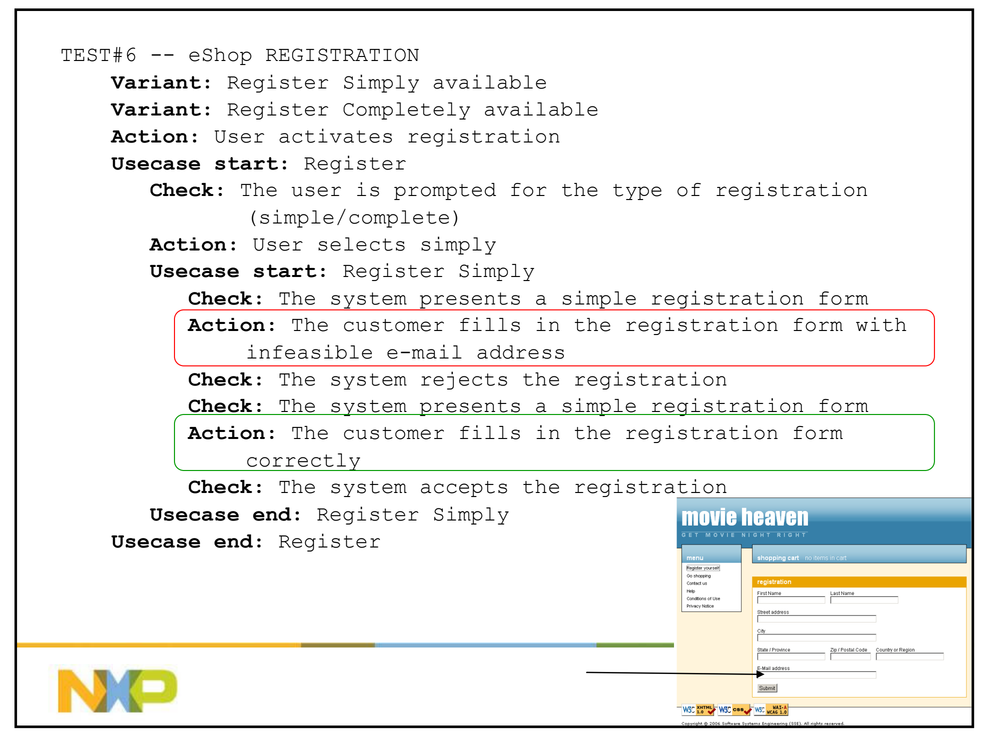
3 problems:

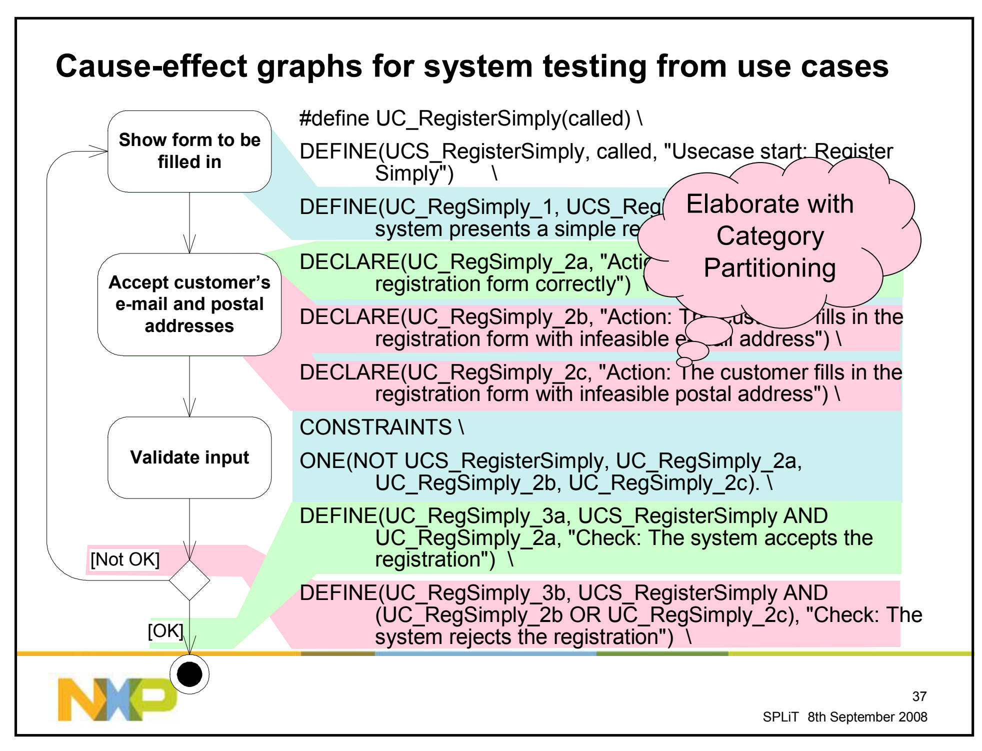
1. Can we find a small set of configurations that make all conditions reachable?
   - Use OAT – probably in the feature model domain (not code level – too many constraints between different code settings)
   - As a component supplier, we *may* have many separate feature models for different customers …
2. For each configuration, can we create a test suite that gives adequate coverage of that configuration?
   - Need highly configurable test suites (easier if abstract) …
   - … that give high levels of coverage (suites need to be detailed)
   - For specification-based testing, to achieve high levels of coverage, the models can become as complex as the code ☹
3. Can our coverage tools accumulate coverage measurements over different configurations?


Automatic installation – Simplified state machine
Pass1
ACI supported
yes
ACIInstallation
no
PreparingFor Pass
Searching
FindingProperties
ACI supported
yes
ACIInstallation
no
VPS/PDC supported
yes
VPSPDCInstallation
no
ACI feature
VPS/PDC feature
InNearFrance feature
ACI supported
yes
StoreACITable
[succeeded]
failed
Installation successful
Pass2
PreparingFor Pass
Searching
FindingProperties
VPS/PDC supported
yes
VPSPDCInstallation
no
31
SPLiT 8th September 2008


Cause-effect graphs for system testing from Use Cases
Show form to be filled in
Accept customer's e-mail and postal addresses
Validate input
[Not OK]
[OK]
movie heaven
GET MOVIE NIGHT RIGHT
menu
Register yourself
Go shopping
Contact us
Help
Conditions of Use
Privacy Notice
shopping cart no items in cart
registration
First Name
Last Name
Street address
City
State / Province
Zip / Postal Code
Country or Region
E-Mail address
Submit
Copyright © 2006 Software Systems Engineering (SSE). All rights reserved.
33
SPLiT 8th September 2008


Register
Register Completely
Register Simply
Show form to be filled in
Show form to be filled in
Show data to customer
[Not OK]
[OK]
Show main page of eShop
Accept customer's e-mail and postal addresses
Validate input
Accept customer's bank account number and sort code
movie heaven
GET MOVIE NIGHT RIGHT
menu
Register yourself
Go shopping
Contact us
Help
Conditions of Use
Privacy Notice
shopping cart no items in cart
registration
Register right now
You can either choose to perform a complete registration (recommended) or a simple registration.
Note: If you choose "simple registration", you will not be able to pay by invoice.
Complete registration
Simple registration
Copyright © 2006 Software Systems Engineering (SSE). All rights reserved.
34
SPLiT 8th September 2008


TEST#6 -- eShop REGISTRATION
Variant: Register Simply available
Variant: Register Completely available
Action: User activates registration
Usecase start: Register
Check: The user is prompted for the type of registration (simple/complete)
Action: User selects simply
Usecase start: Register Simply
Check: The system presents a simple registration form
Action: The customer fills in the registration form with infeasible e-mail address
Check: The system rejects the registration
Check: The system presents a simple registration form
Action: The customer fills in the registration form correctly
Check: The system accepts the registration
Usecase end: Register Simply
Usecase end: Register
movie heaven


Cause-effect graphs for system testing from use cases
Show form to be filled in
Accept customer's e-mail and postal addresses
Validate input
[Not OK]
[OK]
#define UC_RegisterSimply(called) \
DEFINE(UCS_RegisterSimply, called, "Usecase start: Register Simply") \
DEFINE(UC_RegSimply_1, UCS_Reg
system presents a simple re
DECLARE(UC_RegSimply_2a, "Acti
registration form correctly")
DECLARE(UC_RegSimply_2b, "Action:
fills in the
registration form with infeasible e
address") \
DECLARE(UC_RegSimply_2c, "Action: The customer fills in the registration form with infeasible postal address") \
CONSTRAINTS \
ONE(NOT UCS_RegisterSimply, UC_RegSimply_2a, UC_RegSimply_2b, UC_RegSimply_2c). \
DEFINE(UC_RegSimply_3a, UCS_RegisterSimply AND UC_RegSimply_2a, "Check: The system accepts the registration") \
DEFINE(UC_RegSimply_3b, UCS_RegisterSimply AND (UC_RegSimply_2b OR UC_RegSimply_2c), "Check: The system rejects the registration") \
Elaborate with Category Partitioning
37
SPLiT 8th September 2008

Testing during domain engineering

- Am I really going to do this during domain engineering?
  - With my first cu… write test scripts against that co…
  - With change re… ew variation points) that complicate…
  - When a model … difficult to get high coverage with …
- I could use structu… ya Xvantage, VectorCAST, …)
  - They use cons… reachable code
- But the tests aren…
  - Determining th… s writing the test
  - (But they are g… r-preserving refactoring)

38
SPLiT 8th September 2008

Conclusions

- Not having ownership of the final product requirements makes developing a product portfolio for application domains without API standards **much** more difficult than a single product line
- Balancing development effort against test effort is tricky
  - The high observability of the effects of embedded software introduces many more variation points
  - SOA/REST pushes interfaces up a long way from what we want to develop
    - We've been there with the UHAPI
    - Much less of a problem for a product integrator writing all the software
  - We're considering core components + MDA to generate customer-specific behavior
- Achieving high levels of coverage in a highly configurable product portfolio is more difficult than you might think …
  … unless you know differently!

39
SPLiT 8th September 2008

Toward a Fault Model for Software Product Lines

John D. McGregor
School of Computing
Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu

***Abstract*— A software product line organization uses many of the same technologies as any software development project but they are used in a very different context. These differences result in different types of errors being made and different types of faults being present in the software. In addition to different types of errors, the frequency with which the usual types of errors occur will also be different. A fault model provides the basis for constructing effective and efficient test cases since the model describes the faults likely to be present and their relative frequency of occurence. In this position paper we present the initial outline of a fault model for software product lines. We describe the initial steps in the development of the model and discuss how development will continue.**

I. Introduction

How best to test a software product line continues to be the focus of much discussion and research. One of the reasons this remains an open question is the lack of a fundamental model that defines the problem. The problem being what faults are likely to be present in an instance of the product line. The contribution of this paper is to serve as the basis for discussion in the workshop about faults in a software product line.

Munson et al define a fault as "a structural imperfection in a software system that may lead to the system's eventually failing" [1]. We expand that definition for our purposes to be "any imperfection that leads to the system's eventually failing." This provides the basis for searching for faults in the software that are actually caused by errors made in the development of the non-software core assets.

A fault model describes the set of known defects that can result from the activities leading to the test point [2], [3]. Faults can be related to several different aspects of the development environment [4]. For example, programs written in C and C++ are well known for null pointer errors. Aspect-oriented programs also have unique types of faults [5]. Object-oriented design techniques introduce the possibility of certain types of defects such as invoking the incorrect virtual method [6], [7]. Faults also are the result of the development process and even the organization. For example, interface errors are more likely to occur between modules written by teams that are non-co-located.

The development organization develops a fault model for the product line by joining together fault models that reflect the organization's unique blend of processes, domains, and development environment [8], [9]. The organization incorporates existing industry standard models, such as Chillaredge's Orthogonal Defect Classification [10], into the fault models developed for the various test points. Included in this could be fault models for the non-code aspects of the product line organization. The result is a comprehensive fault model for the product line.

Testers use fault models to design effective and efficient test cases since the test cases are specifically designed to search for defects that are likely to be present [11], [12]. Fault models can also help determine how much testing is sufficient to detect specific classes of faults [13]. Developers of safety critical systems construct fault trees as part of a failure analysis. A failure analysis of this type usually starts at the point of failure and works back to find the cause. Another type of analysis is conducted as a forward search. These types of models capture specific faults. We want to capture *fault types* or categories of faults. The models are further divided so that each model corresponds to specific test points in the test process.

The fault model for a software product line is unique in that many of the faults we care about are in the non-software core assets such as the requirements model, plans, or the software architecture. The model addresses these types of faults since the non-software core assets often are the basis for the software core assets. For example, the software architecture defines the interfaces that components must implement. A fault in the architecture may translate into a fault in the software.

Most fault models focus on characteristics of either the language or design paradigm as the source of faults [14]. The software product line approach to building groups of systems does not mandate specific languages or design techniques, but it does affect a wide range of attributes encompassing both engineering and management aspects of the organization. The faults unique to a software product line will be caused by those types of errors related to how the product line organization does business. For example, since most core assets are designed to cover multiple products, the asset often has a more abstract design that is intended to be specialized. This can lead to faults in which the specialized asset is not specific enough or the specialization transform produces an incorrect asset.

In the next section we describe the method used in defining the model. Following that we present the initial model. Finally we discuss plans to continue growing the model.

II. Model Creation Method

There are a number of ways in which a fault model can be created but first we consider how to characterize a fault.

construction	inadequate functionality
disagreements on functionality	changes in functionality
added functionality	misuse of interface
data structure alteration	inadequate error processing
additions to error processing	inadequate post-processing
inadequate interface support	initialization/values errors
violations of data constraints	timing/performance problems
coordination of changes	

TABLE I
INTERFACE FAULT TYPES

language pitfalls	protocol
low level logic	CMS complexity
internal functionality	external functionality
primitives misused	primitives unsupported
change coordination	interface complexity
design and code complexity	error handling
race conditions	performance
resource allocation	dynamic data use
static data design	unknown interactions
unexpected dependencies	concurrent work
other	

TABLE II
GENERAL FAULT TYPES

A. Describing faults

A fault may be characterized by attributes such as when they occur, why they occur, and where they occur. Nikora et al describe several techniques for counting faults [15]. Definitions for counting are useful because they can indirectly define how to identify faults. Munson et al give a technique for identifying a fault from the code deltas in a configuration management system [1]. Essentially the code deltas, which correspond to bug fixes, are analyzed for similarities. This classifies the fixes into categories, which are fault types.

B. Constructing the fault model

A fault model may be created by analyzing the context including application domains, development techniques, and current products for potential sources of faults.

1) Theoretical techniques: Johnson et al demonstrate one approach for fault model creation [16]. They used a theoretical decomposition of potential faults to form the basis of the model. Such a decomposition could be based on a domain analysis. They then used expert judges to classify errors and validate the fault model.

Fault definition during safety analysis uses a fault tree approach [17], [18], [19]. Fault analysis begins at a failure site and backtracks to determine the fault that caused the failure. In [17] this approach is modified and applied to a software product line. In this case, a comprehensive fault tree analysis is created for the product line and then is automatically specialized - the tree is pruned - for each product in the product line [17].

2) Empirical techniques: A product line organization may also develop fault models by tracking defects that are identified during operation of the product line. These defects are classified and defect types emerge. As the model grows it may be refactored and the defect categories structured hierarchically to capture relationships. Orthogonal Defect Classification (OCD) is one example of a classification scheme based on data from real projects [10]. Also, Zoeteweij et al provide a tutorial on fault diagnosis that provides a start on identifying faults [20].

3) Use existing models: There are many fault models that have been created for some specific technique or artifact. These can be used to the extent that they pertain to a software product line. Perry and Evangelist provided a classification scheme for interface faults [21], shown in Table I.

A later paper by Perry and Stieg [22] listed categories for a more general fault model, shown n Table II.

Mutating class attributes	Mutating class variables
Mutating associations	Swap compatible role names

TABLE III
DESIGN FAULTS IN UML MODELS

Dinh-Trong et al developed a taxonomy of faults in UML designs using a mutation analysis approach [23], [24]. They list mutation operators that result in faults. A partial list is shown in Table III.

These classifications are general programming fault models that would apply to the software in a software product line but are not specific to product lines.

For this paper we conducted an extensive literature survey. The survey included all proceedings of SPLiT and other conferences, dissertations and masters theses, and the standard journals. We have not referenced every paper surveyed. The literature was limited to papers that purported to be discussing software product lines, or equivalent approaches under other names. For each paper located in the survey, we searched the paper itself and appropriate references in the paper for specific faults that should be included in the model.

We found two types of papers that were applicable. Some papers implicitly identifed fault types by talking about coverage criteria for defining test cases and describing a rationale for why a particular item was selected to test. Some papers talked explicitly about types of faults that exist in certain domains, types of programs, or with certain designs.

III. FAULT MODEL

First, we consider some of the characteristics of a software product line and its development organization that are the basis for these faults. Then we present a few entries in the fault model. We will also illustrate some of the additional items that will be found in different portions of the core asset base.

A. Characteristics

These characteristics serve as underlying assumptions about, and sometimes constraints on, the model.

- Goal-driven development - A product line organization has explicit business goals and the production method is selected to satisfy those goals. Decisions about models, tools, and techniques impact the likelihood that the business goals will be achieved and, in turn, determine the

types of faults that will be present. The well-definedness of the production method makes it possible to create a sharply focused fault model.
- Distributed development - Development in a software product line is distributed across core asset and product teams, it is often distributed across business units within the organization, and it may be distributed across great geographical distances. Both formal and informal communication suffer as these divides occur. While many syntactic mismatches of interfaces are identified by current software development tools, semantic errors are seldom caught. This places increased emphasis on integration testing. In particular, the fault model should include entries that address various types of interface errors. This refers to human interface as well as software interface issues.
- Variation and configuration - The core assets are intended to be configured for use in a product. The configurability of the assets is directly related to the ability of the assets to provide the range of variability required by the scope of the product line. The many possible configurations of each asset prevent all possible combinations from being tested by the core asset development team. Some of these combinations may result in "emergent behavior," behavior that is not explicit in the specification of either of the assets being integrated but becomes visible in the combination. The fault model should address the errors that are possible for the types of configuration mechanisms used.
- Decision mapping - Individual products are created by the product builder making decisions at numerous variation points. An incorrect or incomplete decision model can result in a product that does not have the required behaviors, and perhaps worse, may have unanticipated behaviors. That is, when a specific parameter value is selected for a variation point, this results in a specific set of modules being added to the product. These modules add specific behaviors to the product whether they are the correct behaviors or not. The fault model should include faults that address a faulty decision making process for technical decisions.
- Constraints - Assets that address the entire scope of the product line may contain inconsistencies. This is acceptable provided that sufficient constraints are included in the model so that products in the product line can only be defined to be internally consistent. When those constraints are missing or incorrect, choices at variation points may result in errors. The fault model contains entries that address regions that are constrained and regions that do not.

B. Variation mechanisms

The variation mechanisms used in software product line design and impementation are a prime source of potential faults. One important characteristic of these mechanisms is the time at which the decision of which variant to use is made. Testing prior to this time requires tests to cover the range of choices, while testing after this time can focus on the selected variant.

Binding times
Source reuse time
Development time
Static code instantiation time
Build time
Package time
Customer customizations
Install time
Startup time
Runtime

TABLE IV
BINDING TIMES

Krueger provides a list of binding times for software product line development [25]. These are shown in Table IV. Krueger's list applies to software core assets. Variability first surfaces in non-software core assets such as the requirements and features models, the product line architecture, and the product line production plan. There are two major binding times for non-software core assets. When the product line version of the asset is created, decisions are made based on the scope of the product line. Values that do not apply to products in the product line are excluded at this point. When the product line template version of the asset is instantiated, choices are made that exclude any values not applicable to the specific product. For example, the product line architecture is created to provide all the possible functional and non-functional requirements. The product line architecture is instantiated for a specific product by choosing among the optional and alternative requirements. It is usual in product line development to have template definition and template instantiation binding times.

C. The model

There are several ways to structure the model. In this initial report we will structure by test point in the testing process. We will treat this as a single model but,in fact, it is really multiple models because each test point has a different purpose, a different set of potential faults, and a different audience. For each test point, the binding timeline can be used to structure the fault types that will be addressed.

A software product line is a set of programs each of which has requirements, and architecture, and design like any program. As such, the faults shown in the various tables apply to the product line, but for now we will not clutter the model with these details. Table VI shows some possible faults related to the product line characteristics in order by test points. Table V gives a beginnngi list of sources for contributions to a comprehensive fault model.

D. Non-software core assets

The Each Asset software product line pattern [26] includes a testing phase for each asset created. This pattern includes assets such as a software architecture for which the Architecture TradeOff Analysis Method is the testing technique [27].

Here is an example set of faults for planning documents.
- Plan is too vague

element	example	potential fault
process models	iterative	inconsistent
programming language	C++	pointer errors
tools	DOORS	bad reference
design paradigm	object-oriented	incorrect use of inheritance
development method	architecture	missing attribute

TABLE V
SOURCES OF FAULT MODELS

- Plan is too detailed
- Plan is incompete
- Plan is not clear

Other types of faults about other types of software product line core assets need to be identified.

IV. EXAMPLE

In this section we present an example of assembling and using fault models in the operation of the pedagogical product line developed by the SEI [28].

The AGM software product line is comprised of nine products, three implementations of three nostalgic games - Brickles, Pong, and Bowling. The product line organization used an incremental approach to building the core assets and products. The first increment was released as freeware, the second as a set of products for wireless devices, and the third was a set of customizable, automatically generated products that incorporates customer specific references such as logos.

The first increment of products was implemented using the C# language while the second and third were implemented in Java. The Java used was the micro-edition for small footprint devices. The switch of language exchanged one set of faults for another; however, since Java and C# are similar many of the same faults are possible in both programming models. This enhanced the reuse of test cases across increments.

A. Creating the model

The fault model for AGM is assembled from several sources. The sources include published fault models for the languages and design paradigms used in the product line. Material for the model also comes from the experiences in the development culture of the organization. Here we detail some other sources:

- By following an incremental approach, AGM visited the various portions of the fault model multiple times. At the start of each increment new requirements, and potentially new development techniques, were added to support the products being developed in the new increment. This introduced faults related to inconsistency among assets, such as the Scope and Business Case documents, as changes are mistakenly only partially propagated. There is also the potential for faults related to transcription errors between successive versions of the same asset.
- The architecture will be evaluated using the ATAM. The architecture of all products follow specific architecture patterns, such as the Model-View-Controller (MVC) pattern. One common fault in the MVC pattern is failure to register a new view with the appropriate controllers.
- The design models were tested against a fault model that would include the model faults given in [23]. The user interface as well as the operation of each game was modeled as a finite state machine. [29] was used to develop standard templates for generating test cases from finite state machines. The interface faults listed in Table I are used to design tests for the architecture description of the component interfaces and then those same faults are used to contribute test cases for the source code. These faults would be added to the Architecture Definition and Detailed Design rows in Table VI.
- Object-oriented languages have a number of fault types. Both C# and Java are object-oriented languages that use virtual machines so they share many fault types. For example, strings are immutable. At times the programmer will use an old string instead of the new one after a modification. Test cases were defined that probed any manipulation of a string. We paid particular attention to array overflows since these account for 85% of security problems in software-intensive products. One fault model for this is given by [30]. McGregor and Sykes provide an association between a number of standard OO design approaches and test case selection [31].
- The test software was written in the same language as the implementation language. This allows the same fault model to be used for testing the tests and the product code. Tests of tests are accomplished via Guided Inspection rather than execution but the test cases are defined in the same way. An implementation of the second increment used aspect-oriented programming to implement several of the variation points. This introduced a specific set of potential faults.

To maximize the level of reuse, the test cases were factored into reusable pieces and then combined to produce test cases for a range of products. A commonality and variability analysis was performed on the requirements for the test software. The implementation of the underlying structure for this approach was written in XVCL. XVCL provides a language in which the tester writes control logic for selecting the appropriate pieces to assemble into complete test cases. The types of faults from this operation are typical of a programming language: incorrectly formed selection statements, incorrect values in variables, and missing options. The binding time for the XVCL control logic is test case construction time. The binding time for XVCL parameters is test case execution time.

B. Using the fault model

The core asset team members are the primary users of the fault model. At each test point in the development process the team must define, construct, and execute a set of test assets including test cases and a test infrastructure.

- Definition - The fault model is used by the core asset team to scope the test cases and the test infrastructure needed at each test point. A core asset team member follows the fault model and defines test cases that are intended to locate any instance of each fault type.

The product development team selects the test cases to be applied at each test point by choosing from the core asset test suite based on variants chosen during product definition.

- Construction - Test software is developed by the core asset team to implement the defined tests. The fault model provides information about the characteristics of the faults, the constraints on those faults, and guides construction.

  The product development team only constructs any unique software needed to supplement the infrastructures and test cases provided by the core asset team.

- Execution - When the test cases are executed, the results are analyzed to determine if any of the failures are the result of faults not anticipated by the fault model. Newly identified faults are added to the model.

  The product development team may be responsible for the analysis of test results looking for new faults or they may only be responsible for providing the raw test results to the core asset team for further analysis. In either case, the core asset team owns the fault model and is responsible for updating it.

The core asset team uses th efault model in Table reffault-Model as a guide to building the core assets needed by the product teams and to identify those testing tasks for which the core asset team is responsible. The product teams use the table to plan the testing conducted during development.

Additional references to other fault models have been added to the table to reflect the actual technologies and tools used by AGM. This includes industry standard fault models such as **fail-stop processors**.

C. Evolving the Fault Model

During the maintenance phase, the fault model has evolved along with the core assets. Defects that are reported are classified and added to the fault model for the test point if they did not exist previously. So far only minor errors and no new fault types have been found.

V. FUTURE

The main purpose of this paper is to provide a context for future work. In this section I will discuss how the fault model might evolve.

In this paper we organized the model by test points. There may be a better structure for the model. We have included information about the test actions and the binding times to aid the tester in decision making. There may be other information that should be added to the model.

The simpler a model the more useful it is. We might be able to impose a higher order structure on the model. Figure 1 shows one possibility using Distribution faults, Variation faults, and Binding faults as the three dimensions.

The variety of types of assets in a software product line will lead to a model that is an aggregation of models. Each model can be developed separately. Some of the models may already exist. Any new faults identified during the operation of the product line should be examined to determine if they open a new area that should have its own model.

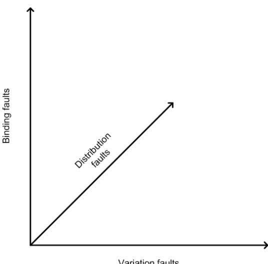


Fig. 1. Possible model structure

There are many flavors of product line practice. To make this model as widely applicable as possible, each potential fault should be accompanied by assumptions and constraints that characterize the type of product line in which the fault is likely to occur.

Testers are often too concerned about failures to pay attention to the faults that cause the failures. Part of the development of the fault model should be communication of the model. One possible approach is an open approach to the development of fault models. We are considering using this basic model as the initial contribution for a new open creation effort hosted on one of the open source infrastructures.

VI. CONCLUSION

We have presented an initial fault model for software product lines. The model describes faults that are the result of applying the product line strategy. The model is intended to be used along with basic fault models for the design and implementation techniques chosen to be a part of the production method.

The model we have presented is incomplete. We hope that the community, particularly the SPLiT community, will participate in identifying classifications and specific faults that will shape the model.

REFERENCES

[1] J. C. Munson, A. P. Nikora, and J. S. Sherif, “Software faults: A quantifiable definition,” *Advances in Engineering Software*, vol. 37, pp. 327–333, 2006.

[2] L. J. Morrell, “A theory of fault-based testing,” *IEEE Transactions of Software Engineering*, vol. 16, 1990.

[3] J. Pan, P. Koopman, and D. Siewiorek, “A dimensionality model approach to testing and improving software robustness,” in *Proceedings of Autotestcon 99*, 1999.

[4] M. Esser and P.Struss, “Fault-model-based test generation for embedded software,” *IJCAI*, 2007.

[5] J. S. Baekken and R. T. Alexander, “A candidate fault model for aspectj pointcuts,” in *17th International Symposium on Software Reliability Engineering*, 2006.

[6] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, “A fault model for subtype inheritance and polymorphism,” in *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE ’01)*, 2001, pp. 84–95.

AGM Fault Model				
Test point	**Targeted faults**	**Constraints**	**Binding Time**	**Test Actions**
Requirements	Failure to cover the full scope	Iterative development	Product line scoping time	Guided Inspection
		increases risk		
	Incomplete list of variations			
	Failure to capture required qualities			
Analysis	Missing constraints on variations			
	Incorrect constraints on variations			
	Unclear definition of boundary between			
	common and variable behavior			
Architecture design	Contradictions between variation constraints			
	Failure to accommodate full range of variability			
	Failure to achieve quality attribute goals			
	Failure to identify architectural variability			
	Java MicroEdition design pattern violations	small footprint	architecture definition	ATAM
Detailed design	Failure to propagate variations	check visibility rules	architecture definition	Guided Inspection
	from subsystem interface			
	Inappropriate variant binding time			
	Failure to provide sufficient details			
	across the scope of the product line			
Unit testing	Failure to implement expected variations			
	Java programming fault model	resolve external all dependencies	programming time	JUnit tests execution
Integration testing	Mismatched binding times between modules	see Table I		
	Mismatched types in provided/requires interfaces	see Table I	detailed design time	
	Fail stop processors fault model [32]	assumes multi-threading	detailed design time	SyncPath testing[11]
System testing	Inability to achieve required configuration			
	Usability fault model		runtime configurable	TPTP GUI test suite

TABLE VI
FAULTS BY TEST POINT

[7] R. T. Alexander, J. Offutt, and J. M. Bieman, "Syntactic fault patterns in oo programs," in *Eighth International Conference on Engineering of Complex Computer Software (ICECCS '02)*, 2002.

[8] M. A. Sanchez, J. C. Augusto, and M. Felder, "Fault-based testing of e-commerce applications," in *Proceedings of VVEIS*, 2004.

[9] W. Chan, S. C. Cheung, and T. Tse, "Fault-based testing of database application programs with conceptual data model," in *Proceedings of the 5th International Conference on Quality Software (QSIC 2005)*, 2005.

[10] R. Chillarege, "Odc crystallizes test effectiveness," in *Proceedings of SPLiT - The Third Annual International Workshop on Testing Software Product Lines*, 2006, pp. 31–32.

[11] K.-C. Tai, "Theory of fault-based predicate testing for computer programs," *IEEE Transactions of Software Engineering*, vol. 22, 1996.

[12] I. Lee, R. K. Iyer, and A. Mehta, "Identifying software problems using symptoms," in *Twenty-Fourth International Symposium on Fault-Tolerant Computing, 1994. FTCS-24*, 1994.

[13] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[14] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," in *Proceedings of ISSTA'07*, 2007.

[15] A. P. Nikora and J. C. Munson, "The effects of fault counting methods on fault model quality," in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC04)*, 2004.

[16] H. L. Johnson, K. B. Cohen, and L. Hunter, "A fault model for ontology mapping, alignment, and linking system," in *Pacific Symposium on Biocomputing*, 2007, pp. 233–268.

[17] J. Dehlinger and R. Lutz, "Plfaultcat: A product-line software fault tree analysis tool," in *Proceedings of Automated Software Engineering*, 2006.

[18] J. Dehlinger, M. Humphrey, L. Suvorov, P. Padmanabhan, and R. Lutz, "Decimal and plfaultcat: From product-line requirements to product-line member software fault trees," in *Proceedings of the International Conference on Software Engineering*, 2007.

[19] D. Lu and R. R. Lutz, "Fault contribution trees for product families," in *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002.

[20] P. Zoeteweij, R. Abreu, and A. J. van Gemund, "Software fault diagnosis," 2007, presented at TESTCOM/FATES 2007.

[21] D. E. Perry and W. Evangelist, "An empirical study of software interface faults," in *Proceedings of the International Conference on Software Engineering*, 1985.

[22] D. E. Perry and C. S. Stieg, "Software faults in evolving a large, real-time system: a case study," in *Proceedings of the Fourth European Software Engineering Conference*, 1990.

[23] T. Dinh-Trong, S. Ghosh, and R. B. France, "A systematic approach to generate inputs to test uml design models," in *17th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2006.

[24] ——, "A taxonomy of faults for uml designs," in *Second Model Design and Validation Workshop*, 2005.

[25] C. W. Krueger, "Product line binding times: What you dont know can hurt you," in *Proceedings of the Software Product Line Conference, 2004*. Springer, 2004.

[26] P. Clements and L. M. Northrop, *Software Product Lines : Practices and Patterns*, ser. Professional. Addison-Wesley, 2002.

[27] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Publishing, 2001.

[28] Sei pedagogical product line. [Online]. Available: http://www.sei.cmu.edu/productlines/pedagogical

[29] A. Petrenko, "Fault model-driven test derivation from finite state models: Annotated bibliography," in *Modeling and Verification of Parallel Processes*, ser. Lecture Notes in Computer Science, 2001.

[30] C. Gaoa, M. Duana, L. Tana, and Y. Gongb, "Out-of-bounds array access fault model and automatic testing method study," *Tsinghua Science and Technology*, vol. 12, pp. 14–19, 2007.

[31] J. McGregor and D. Sykes, *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley Professional, 2001.

[32] R. D. Schlichting and F. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, p. 222238, 1983.

Validation of the Reliability Analysis Method and Tool

Anne Immonen and Antti Evesti
VTT Technical Research Centre of Finland
P.O. Box 1100 FIN-90571 Oulu, Finland
{anne.immonen, antti.evesti}@vtt.fi

Abstract

The high reliability of today's software systems is mandatory as these systems are required to work as intended without fault. Reliability problems detected after system implementation are difficult to deal with, and therefore the present day tendency has been to evaluate the reliability of software at an early phase. The RAP (Reliability and Availability Prediction) method and tool have been developed to assist software developers in analyzing reliability and availability at the architectural level, before system implementation. Our contribution is the refinement and validation of the quantitative analysis part of the RAP method and tool. The analysis method and the tool have been improved upon from earlier versions to make analysis as more fluent part of software development. The validation is implemented with the case study.

1. Introduction

Reliability is part of the general quality of software, and can be defined as a probability of failure-free operation of a software system for a specified period of time in a specified environment [10]. Reliability evaluation at an early stage makes system modifications easier and cheaper and save time in later development phases. From an architecture point of view, reliability is one of the execution qualities of a software system [2]. Software architecture is the first asset that describes the software system as a whole, and therefore it can be used as a starting point for a predictive reliability analysis. However, the analysis using the architecture is only possible if the architecture is represented in a way that enables analysis [7]. There exist several reliability analysis methods in the literature that focus on analyzing reliability at the architectural level. The most promising of them from the software architecture viewpoint are surveyed and compared in [6]. Due to the identified shortcomings of the existing analysis methods, the RAP method [4] was developed. The method was meant to be used in product family-based software development, assisting to enclose reliability into family and system architecture, and to analyze whether the architecture meets the reliability requirements. The RAP tool [5] was developed to assist in the quantitative reliability analysis of the RAP method.

Reliability analysis is challenging for several reasons. Systems are often complicated and distributed to several platforms over networks. The success of the analysis requires a comprehensive knowledge of the system. Some of the knowledge can be tacit, or the knowledge comes from the experience of the designer, while some of it the designer derives from the system requirements. Therefore, the reliability analysis should be a part of the typical activities of software architects. Since architecture design and analysis are iterative processes, the analysis should be able to be performed quickly. Software developers need a clear method and tools to facilitate and automate the reliability analysis of a software system and the involved components. Although several reliability analysis methods exist, there are no common practices on how to implement analysis or even how to model architecture for the analysis. Furthermore, there is seldom tool support for the methods.

The main purpose of this paper is to validate the use of the RAP method and RAP tool in the reliability analysis of a case study. The RAP method is three-phased, containing separate phases for the requirements definition, architecture modelling and evaluation; the validation in this paper concentrates only on the quantitative reliability analysis of the evaluation phase. This phase has been modified from the earlier version to correspond better to the requirements for today's reliability analysis, e.g. the architect's effort for the analysis has been diminished. The case study is the Personal Information Repository (PIR) document

service [8] that mediates health care information between health care professionals and customers.

We have also a plan to provide dynamic testing support as a part of the RAP tool. ComponentBee testing tool [11] will provide this support after tool implementation is ready. ComponentBee can produce reliability values for the implemented components and RAP tool will utilise this data further. However, this part is out of scope of this work.

This paper is organized as follows: Chapters 2 describes our reliability analysis method and the updated tool. The validation of the method and tool is described in Chapter 4, and finally the conclusions are drawn in Chapter 5.

2. The reliability analysis method and tool

The reliability analysis method described here covers the quantitative analysis part of the RAP method [4]. The most significant shortcoming of the method was that it still required quite an effort from the software architect. In addition to a general architectural description, the method required a special simulation model and input messages for the analysis. The main idea of the improved reliability analysis method is that the analysis can be done with the already existing architectural models. The simulation model and input messages are no longer required, and the method is based on usage profiles and system execution paths. The usage profiles are also closely linked to the reliability of components. The usage stresses components, and therefore, the more the component is used, the more likely its probability of failure increases [3]. The purpose of the usage profile description is to define all the system execution possibilities and their probabilities that are further used in component and system level behavioural descriptions.

The analysis method requires the following architectural descriptions as inputs: a behavioural description of the components, and structural and a behavioural description of the system. The structural description is modelled with the help of a component diagram. The behaviour is described using the sequence diagrams that describe component interactions when executing system tasks. Each sequence diagram describes one system execution path. The execution paths are used when analyzing system level reliability.

Component behaviour is described using a state diagram. The state diagram is transformed to the Markov chain model [1] by adding state transition probabilities and a reliability value for each state. This information is based on the architect's knowledge which he can derive from the component's requirements specification. The Markov model is used to calculate the component reliability as an independent unit, separated from the system architecture. This value is further used in the system level reliability analysis. It is not always rational to define Markov models, e.g. for small or simple components. In such cases it is enough that the architect gives the estimated probability of failure value for the component.

It is not always possible to analyze the entire system. Different usage profiles can be analyzed separately by selecting system execution paths involving a certain usage of the system. The analysis is implemented as path-based, basing itself on the selected system execution paths and their execution probabilities. The values for the probability of failure of components are needed for this kind of analysis. The analysis reveals the probability of failure of each system execution path, and in addition, the system dependent reliability for each component. Finally the system reliability is calculated as a weighted average of the probability of failure of all the system execution paths. As summary, definitions for different probability of failure values are:

- Estimated probability of failure is based on architect's experience or reference values got from the component provider. Alternatively the Markov models can be used.
- Predicted probability of failure values are calculated by the RAP tool.
- Measured probability of failure values are produced from running system or component by ComponentBee.

RAP tool: The first version of the RAP tool [5] was implemented in C#. Instead, the new version of the tool is implemented with Java, utilizing the Eclipse platform. The architecture of the RAP tool obeys the Blackboard architectural pattern (Figure 1). The biggest changes to the earlier RAP tool version are: 1) the analysis tool offers the possibility of changing the UML [9] compliant modelling tool, 2) the different usage profiles can be defined and analyzed easily, and 3) the effort of the architect is diminished considerably.

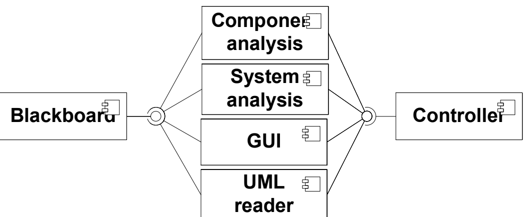


Figure 1 Architecture of the RAP tool.

The component reliability analysis can be done utilizing the component's state diagram by changing it

to the Markov model. For a system wide reliability analysis, the tool requires the system execution paths in the form of sequence diagrams and the path execution probabilities based on the selected usage profile. The tool calculates the predicted probability of failure values for each component in the system (i.e. system dependent reliability) and the predicted probability of failure value for the whole system. Figure 2 shows an overview of the analysis. The ComponentBee reliability testing tool is also included in the picture because it is an essential part when verifying reliability from the running system or component.

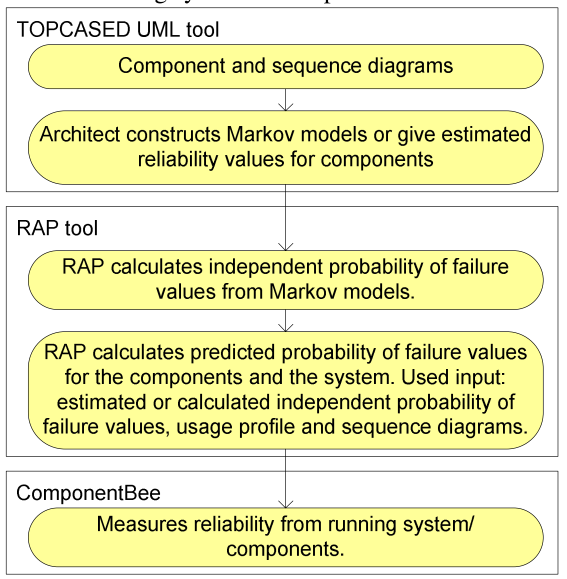


Figure 2 Overview of the RAP tool's usage.

The RAP tool provides user interfaces for defining the inputs for analysis. In the system analysis window the user can select the sequence diagrams for the analysis and define the execution times for the system path execution (Figure 5). The user sees the results of the analysis in an analysis results window (Figure 6 and Figure 7).

3. Applying the method and tool to a case study

3.1. Description of the case study

The reliability analysis method and the tool were validated by using them in a reliability analysis of a case study; the Personal Information Repository (PIR) document service [8] (http://pir.fi/eng/index.html). The PIR system implements a business-to-consumer document delivery system, providing an architecture that enables the sharing of services with several business to consumer service providers and allowing customers to use the same interface in communicating with several service providers.

The main purpose of the PIR system (see Figure 3) is to mediate health care information between hospitals (i.e. service providers) and customers (i.e. patients). Data is encrypted and transmitted using the HTTPS protocol. This data can be messages, documents e.g. Word, special forms filled with the Formfiller application or graphs drawn with the CDAGraph application. The server component of the system acts as a temporal data store, mediating documents between customers and service providers. The upload component assists in uploading and deleting the documents from the server. Health care professionals enter the PIR system using the PIR Mailer client component. The component offers the basic document exchange and management functionalities. The customers enter the system using either PIR browser or PIR case client components. Both components include functionalities for sending, receiving and managing documents, while locally installed PIR case software also provides support for local document management and encryption. The customers are identified using outsourced authentication services – Tupas - offered by a bank. The document receiver is notified about the incoming document with e-mail or SMS messages. The CIA component takes care of establishing and maintaining connections between the service providers and their customers. It also takes care of sending e-mail and SMS notifying.

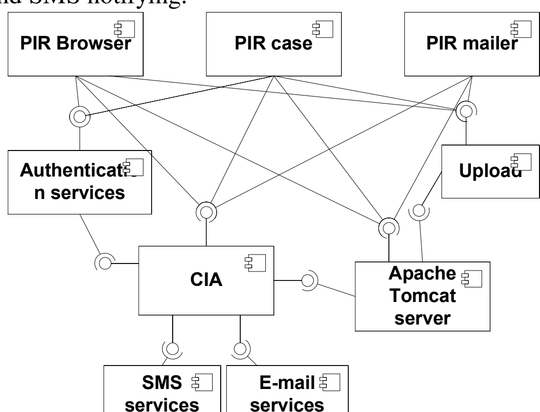


Figure 3 Structural description of the PIR architecture.

High reliability of the PIR system is required, since the system includes confidential data. Therefore, the system probability of failure must be 0; information may not be lost, corrupted or be accessed by unauthorized users. The PIR system has been validated in a user trial in Finnish hospitals and health centres. At the moment the PIR system is in pilot use in one hospital and two health centres in Finland. The

experiments from the pilot use are positive; there would be demand for a PIR system more broadly in hospitals and health centres.

3.2. Reliability analysis of the case study

We interviewed two architects of the PIR system for collecting information required to perform the analysis. The interview took approximately three hours and the main outcome from it was component and sequence diagrams, estimated reliability value for each component and the Markov models for two main components. In addition, architects gave the estimated execution times for each sequence diagram. When the architects were asked to do the reliability analysis based on their personal knowledge about the system, the following findings came out: The server side of the system was estimated to be more reliable than the client side. On the client side, the local states of the components were estimated as being more reliable, e.g. when viewing a document locally. Formfiller, CDAGraph and PIR mailer DB components were estimated to be the most unreliable components of the system. E-mail service and Tupas components were estimated to be the most reliable ones. Table 1 shows the main components of the system and their types and the estimated probability of failure values. Apache Tomcat server is an OS component but it also contains self-made additions. Thus, its estimated probability of failure value is in the same level as other self-made components have.

Table 1 Type and estimated probability of failure (pof) for the main components.

Component	Type	Estimated pof
PIR browser	Proprietary	0.0050
PIR mailer	Proprietary	0.0080
PIR case	Proprietary	0.0060
PIR mailer DB	Proprietary	0.0120
Upload	Proprietary	0.0030
Tupas	Commercial	0.0000
Formfiller	Proprietary	0.0100
CDAGraph	Proprietary	0.0100
SMS-services	Proprietary	0.0100
CIA	Proprietary	0.0030
E-mail service	OS	0.0010
Apache Tomcat server	OS + additional code	0.0030

Component reliability analysis: When defining a Markov model, the architect had to define state transition probabilities and probability of failure values for each state (see Figure 4). For instance, the Markov model in Figure 4 gives 0.0040 probability of failure for the PIR case component. The PIR mailer component has also the Markov model, which gives 0.0040 probability of failure value. By comparison, the Markov model gives 33% smaller probability of failure value for the PIR case and 50% smaller for the PIR mailer than the first estimate (see Table 1).

System reliability analysis: For the system analysis, the architect had to select the sequence diagrams and define the path execution probabilities. The execution probabilities are given in the form of path execution times in relation to other paths (see Figure 5).

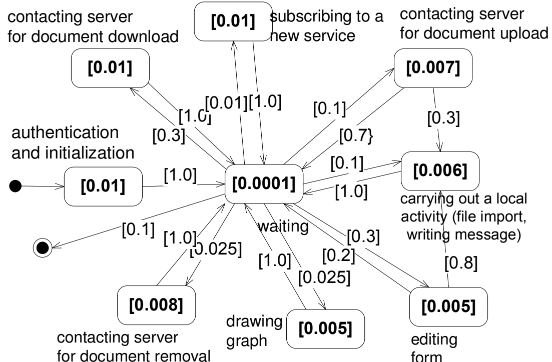


Figure 4 Markov chain model of the PIR case component.

The results of the analysis are shown in the "analysis results" window for both components and the system. The component window (Figure 6) shows the probability of failure value of a component as an independent unit and the system dependent unit, and the component's execution times. According to the probability of failure of independent components, the outsourced authentication service Tupas and the open source e-mail service seemed to be the most reliable and the PIR mailer DB the most unreliable one. Correspondingly, the most reliable component in the system execution is Tupas and the most unreliable are PIR mailer DB and Apache Tomcat.

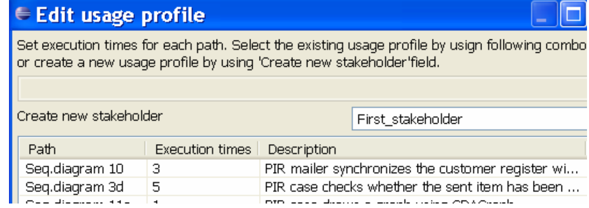
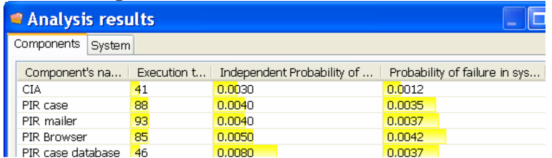


Figure 5 Paths' execution probabilities.

The outcome of the analysis is surprising regarding the result of the Apache Tomcat component. The Apache Tomcat component is the most executed component as a whole. However, it is not an adequate reason for the result. Instead, self-made code added to the component raises the estimated probability of failure value. Together the high values of execution probability and the estimated probability of failure

cause an elevated probability of failure for the Apache Tomcat component.



Analysis results

Components | System

Component's na...	Execution t...	Independent Probability of ...	Probability of failure in sys...
CIA	41	0.0030	0.0012
PIR case	88	0.0040	0.0035
PIR mailer	93	0.0040	0.0037
PIR Browser	85	0.0050	0.0042
PIR case database	46	0.0080	0.0037

Figure 6 Results of the component analysis.

The system analysis window (see Figure 7) shows the result of the analysis for the whole system. The predicted probability of failure of the system was calculated to be 0.042647. Thus, the system could fail with the probability of 4.3%. The execution paths, path execution times, and paths' probability of failure can be detected from the system analysis window.

Analysis results

Components | System

System's probability of failure is 0.042647

Path's name	Execution ti...	Probability of ...	Description	Components in path
Seq.diagram	3	0.0218	PIR mailer synchronizes the customer register with database	Cuver PIR mailer database PIR mailer
Seq.diagram	5	0.0267	PIR case checks whether the sent item has been	Apache Tomcat PIR case

Figure 7 Results of the final analysis.

When the architects of the PIR system compared the final result 4.3% with their own experience of the implemented PIR system they noticed that the predicted value was higher than they had expected. The main reason for this difference can be found from the estimated probability of failure values. These values contain uncertain and pessimistic values because some components of the PIR system are under construction and, therefore, these components got the higher estimate than really needed. Moreover, only two components had the Markov models and difference between the values calculated from the Markov model and the estimated values were considerable. Thus, the predicted probability of failure value for the PIR system contains uncertainty.

3.3. Lessons learned

The main lesson learned from the analysis was that when effort has been made on specifying the architecture carefully, the analysis itself is not time consuming or difficult. The analysis of the PIR architecture could be done quickly because the additional simulation model is not needed anymore. Instead the analysis can be performed from the common architectural descriptions, i.e. UML diagrams. Usage profiles seem to be a reasonable way to depict different usages of the system that facilitates to analyse system from different user perspectives.

We remarked that giving an estimated probability of failure values is a difficult task as such. Thus, it is important to define the Markov model for frequently used components because there was significant difference between the estimated value and the value calculated from the Markov model. However, if estimated probability of failure values are used it is important to estimate the component's final reliability, instead of taking into account the development time faults that will be fixed during the development. These issues have a strong influence to the final results.

Conclusions from the PIR analysis have to make carefully because of uncertainty in the estimated probability of failure values. Table 2 contains same components as Table 1 with analysis results. As the analysis using the RAP method revealed, the client side, especially the PIR mailer DB, was attested to be the most unreliable. The Formfiller and CDAGraph components were estimated to be the most unreliable components as such, but the probability of failure values of these components in the actual use within the system was not considered to be weak, especially in the case of the CDAGraph. The most used and therefore critical components of the system were found to be the Apache Tomcat server component and the Upload component on the server side. Therefore, special attention should be paid to how to ensure the reliability of these components. On the client side, the most frequently used components were PIR mailer, PIR case and PIR browser components.

Table 2 Predicted probability of failure (pof).

Component	Execution times	Predicted pof
PIR browser	85	0.0042
PIR mailer	93	0.0037
PIR case	88	0.0035
PIR mailer DB	52	0.0062
Upload	120	0.0036
Tupas	2	0.0000
Formfiller	45	0.0045
CDAGraph	4	0.0004
SMS-services	30	0.0030
CIA	41	0.0012
E-mail service	30	0.0003
Apache Tomcat server	198	0.0059

It is up to the architect to decide whether to accept the architecture or to effect some changes. One of the requirements was that the system probability of failure value must be 0. Therefore, the architect decides whether or not the achieved value through the analysis, 0.042647, is close enough. The architects of the PIR system did not agree with this value and their experience from the implemented system proved that the value is too high. Therefore, this result enforces to

pay particular attention to how to define these estimated probability of failure values in the future.

4. Conclusions

This paper introduced a validation of the evaluation part of the RAP method and its supporting tool. The case study used was the PIR document service, which is a business-to-consumer document delivery system developed for the health care domain.

The validation performed in this paper concentrated on the improved quantitative reliability analysis method that is a part of the reliability evaluation phase of the RAP method. The existing RAP tool had been improved to automate the analysis work further. Experiences with the analysis method and tool showed that they enabled quick and easy architectural level reliability analysis for both components and the system. The use of the method and tool did not require any special user skills, and the analysis could be made using the basic architectural descriptions as inputs. The architect no longer has to construct a separate simulation model and input messages. Instead, commonly used sequence diagrams replace this information. This enhancement facilitates analysis process to an outstanding degree. The validation also exposes that defining the estimated probability of failure values is a difficult task and may cause a considerable deviation to the final results. Thus, it is important to define the Markov model when ever possible.

The further development of the method and tool is work in process for the authors of this paper. Development targets include, for example, support for system distribution, and the calculation of other reliability metrics. Furthermore, the purpose is to realize the co-operation between the analysis tool and the ComponentBee reliability testing tool, as soon as the work on the testing tool has been completed. ComponentBee makes it possible to compare predicted probability of failure values to the measured values. This will be an important feature when verifying analysis results in the future.

5. Acknowledgments

This work was carried out at the VTT Technical Research Centre of Finland, within the Eureka ITEA research project COSI. We would like to thank Mr. Jaakko Lähteenmäki, Mr. Juha Leppänen, Mr. Hannu Kaijanranta and Mr. Juha Sjöholm from the VTT Technical Research Centre of Finland for providing the case study.

6. References

[1] R.C. Cheung, A user-oriented software reliability model. IEEE Transactions on Software Engineering, 1980 Vol. 6 No 2, pp. 118-125.

[2] J.C. Dueñas, W. de Oliveira and J. de la Puente, A software architecture evaluation model, Proceedings of the second international ESPRIT ARES workshop on development and evolution of software architecture for product families. Las Palmas de Gran Canaria Spain, 26 - 27 February 1998, Springer-Verlag, Berlin Germany, 1998, pp. 148 - 157.

[3] W. Everett, Software component reliability analysis, IEEE Symposium on Application - Specific Systems and Software Engineering and Technology, Richardson, Texas, 1999, pp. 204 – 211.

[4] A. Immonen, A method for predicting reliability and availability at the architectural level, in Software Product-Lines - Research Issues in Engineering and Management, T. Käkölä and J.C. Dueñas, Editors, Springer-Verlag: Berlin Heidelberg 2006, pp. 373-422.

[5] A. Immonen and A. Niskanen, A tool for reliability and availability prediction, Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications, IEEE Computer Society, Porto Portugal, 2005, pp. 416 - 423.

[6] A. Immonen and E. Niemelä, Survey of Reliability and Availability Prediction Methods from the Viewpoint of Software Architecture, Software and Systems Modeling, Springer Verlag, Heidelberg, 2008 Vol. 7. No 1, pp. 49 – 65.

[7] M. Jazayeri, A. Ran and F. van der Linden, Software Architecture for Product Families, Addison-Wesley, Boston USA, 2000.

[8] J. Lähteenmäki, J. Leppänen and H. Kaijanranta, Document-based service architecture for communication between health and wellness service providers and customers, accepted to 2nd International Conference on Pervasive Computing Technologies for Healthcare, 2008.

[9] OMG, Unified Modeling Language (UML), version 2.0, Object Management Group, 2005.

[10] R.H. Reussner, H.W. Schmidt and I.H. Poernomo, Reliability prediction for component-based software architectures, Journal of Systems and Software, Elsevier. USA, Vol. 66 No 3 2003, pp. 241 - 252.

[11] VTT, ComponentBee, URL: http://www.vtt.fi/proj/cosi/cosi_ComponentBEE.jsp.

Functional Testing of Feature Model Analysis Tools. A First Step *

Sergio Segura, David Benavides and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
University of Seville
Av Reina Mercedes S/N, 41012 Seville, Spain
{sergiosegura, benavides, aruiz} AT us.es

Abstract

The automated analysis of Feature Models (FMs) focuses on the usage of different logic paradigms and solvers to implement a number of analysis operations on FMs. The implementation of these operations using a specific solver is an error-prone and time-consuming task. To improve this situation, we propose to design a generic set of test cases to verify the functionality and correctness of the tools for the automated analysis of FMs. These test cases would help to improve the reliability of the existing tools while reducing the time needed to develop new ones. As a starting point, in this position paper we overview some of the classifications of software testing methods reported in the literature and study the adequacy of each approach to the context of our proposal.

1. Introduction

The analysis of an FM consists on the observation of its properties. Typical operations of analysis allow finding out whether a FM is void (i.e. it represents no products), whether it contains errors (e.g. feature that can not be part of any products) or what is the number of products of the software product line represented by the model. In particular, the analysis is generally performed in two steps: i) First, the model is translated into a specific logic representation (e.g. Constraint Satisfaction Problem (CSP), [4]), ii) Then, off-the-shelf solvers are used to automatically perform a set of analysis operations on the logic representation of the model [5].

The available empirical results [6, 7] and surveys [5] in the context of the automated analysis of FMs suggest that there is neither an optimum logic paradigm nor solver to perform all the operations identified on FMs. As a result of this, many authors propose enabling the analysis using different paradigms such as constraint programming [4], propositional logic [3, 16] or description logic [9, 15].

Implementing the operations for the analysis of FMs using a specific solver is not a trivial task. The lack of specific testing mechanisms in this context difficult the development of tools and reduce their reliability. To improve this situation, we propose to design a set of generic test cases to verify the functionality and correctness of the tools for the automated analysis of FMs. These test cases would help to improve the reliability and robustness of the existing tools while reducing the time needed to develop new ones. As a starting point, in this position paper we narrow the search for a suitable testing technique, test adequacy criteria and test data generation mechanism to be used for our tests. For that purpose, we overview some of the classification of software testing methods reported in the literature and study the adequacy of each approach to the context of our problem.

The remainder of the paper is structured as follows: In Section 2 we introduce some common classification of testing techniques and evaluate the adequacy of each approach for our purposes. Some general classes of test adequacy criteria and some argumentation about whether they are appropriate for our proposal are presented in Section 3. In Section 4 we study different mechanisms for the generation of test data. Finally, we overview our evaluation of the different approaches and describe our future work in Section 5.

2. Selection of testing techniques

Testing techniques can be classified according to multiple factors. Next, we describe some of them and evaluate the adequacy of each approach to the context of our proposal.

- **Knowledge of the source code.** According to our knowledge about the source code of the program un-

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533)

der test, tests can be classify as *white-box*, *black-box* or *grey-box* test cases [8, 10, 11, 12, 17]. Black box test cases are those in which no knowledge about the implementation is assumed. These test cases are based exclusively on the inputs and outputs of the system under test. White-box (or, alternatively, *glass-box*) test cases consider the entire source code of the program while grey-box test cases only consider a portion of it.

We want our proposal to be suitable to test any tool for the automated analysis of FMs independently of the logic paradigm or solver it uses for the analysis. Therefore, we will need a black box testing technique in which only the input and output of the tools are considered.

- **Source of information used.** According to the source of information used to specify testing requirement, testing techniques can be mainly classified as *program-based*, *specification-based* and *interface-based* [8, 10, 17]. Program-based testing approaches base on the source code of the program to create the tests. Specification-based techniques based on the specification (i.e. requirements) of the program to identify features to test. Finally, interface-based test cases specify testing requirements in terms of the type and range of the inputs without referencing any detail of the program or the specification.

  Since we have decided to used black box testing techniques we can only consider the approaches not assuming any knowledge about the source code, that is, specification- and interface-based testing. On the one hand, specification-based testing appears as a suitable strategy since we assume that the tools for the automated analysis of FMs share a common functional specification for the analysis operations. On the other hand, we presume interface-based is not suitable for our proposal since it only deals with the information provided by the interface without considering any functional aspect of the software under test.

- **Execution-based vs Non-execution based.** Testing can be perform either by running the program to test or without running it through the usage of so-called *software inspections* [10, 12]. Non-execution based methods usually require to examine the source code of the program and the documentation that accompanies it. Thus, we consider that an execution-based technique is probably the best option to our purpose since we do not assume any knowledge about the source code.

- **Testing level.** Testing can be performed at different levels, mainly: *unit*, *integration* and *system level* [2]. Unit testing focus on the verification of isolated component or modules. Integration testing exposes defects in the interfaces and interaction between integrated modules. Finally, system testing test the whole system to verify whether it fulfill the requirements or not.

  We intend to verify the functionality of each analysis operation separately in order to be as accurate as possible when informing about defects. Consequently, we consider that unit tests are the most suitable approach for our needs.

3. Selection of test adequacy criteria

Adequacy criteria define what constitutes a good test. Zhu *et al.* [17] identify three generic classes of test adequacy criteria. Next, we introduce them and check whether they are appropriate for our context of application.

- **Fault-based.** This criterion measures the adequacy of a test according to its ability to detect faults. Typical strategies using this criterion are *error seeding* or *program mutation testing* [17]. In these approaches, artificial faults are introduced into a copy of the program (or one of the programs) under test. Then, the software is tested and the number of artificial faults detected is counted. The proportion of artificial faults detected determines the adequacy of the test.

  We consider that the usage of this kind of adequacy criterion would be suitable for our proposal since the main goal of our tests is to expose defects in the tools for the automated analysis of FMs. For instance, once the test were ready, several mock implementations of the analysis operations including artificial faults could be developed and used to measure the ability (i.e. adequacy) of our tests to find them.

- **Error-based.** Error-based testing requires test cases to check the software on certain error-prone points [17]. This way, this criterion measures the ability of the tests to verify that the software does not deviate from its specification in a typical way.

  Our experience in the analysis of FMs has leaded us to identify several error-prone points when designing tools for the automated analysis of FMs. These are especially related with the usage of feature cardinalities and the detection of dead features [13]. Thus, we consider this adequacy criterion is also a suitable option to take into account for our purpose.

- **Structurally-based.** A structurally-based criterion requires the design of a test that covers a particular set of elements (e.g. statements) in the structure of the program or the specification. This test adequacy criterion is mostly used with program-based testing [10]. Thus, we do not find it particularly useful for our tests.

4. Selection of mechanisms for test data generation

We identify different options to generate the test data, namely:

- **Automated *vs* manual.** The generation of test data can be either manual or automated [10]. On the one hand, the automated generation of test data is faster and usually enables the generation of bigger and more complex program inputs. On the other hand, the manual generation simplifies the creation of customized tests.

  We consider that both approaches are suitable for our approach and that they could even being combined. This way, the test inputs could be composed by two groups of automatically- and manually-generated FMs.

- **Random *vs* systematic.** Test data can be generated either randomly or systematically. On the one hand, random test data generation approaches relies upon a random number generator to generate test input values [10]. On the other hand, systematic approaches follow some pre-defined criteria to generate the data.

  Once again, we consider that a combined approach would be the most suitable option for our tests. For instance, a set of randomly-generated FMs could be first used to test the tools using different size and forms of the input. Then, a second group of FMs could be systematically designed to exercise some specific error-prone points in the tools under test. In this context, we already have a tool, the FAMA[1] framework [14], providing support for the automated generation of random FMs.

5. Overview and open issues

Figure 1 illustrates an FM summarizing the factors (i.e. features) we considered for the selection of an adequate testing techniques, test adequacy criteria and test data generation mechanism. Cross-tree constraints are not included for simplicity. The studied features are mainly based on the decomposition of different testing techniques and adequacy criteria proposed by Binder [8], Kapfhammer [10] and Zhu et al. [17]. However, we remark that the usage of other classifications of software testing methods could also be feasible.

Filled features in Figure 1 illustrate the set of configurations that we evaluated as adequate for our proposal. These configurations helped us to refine our contribution and motivated some of the main research questions to be addressed in our future work, namely:

[1] http://www.isa.us.es/fama

- **What specific technique should be used to define the tests?** We concluded that we should use a black box, execution- and specification-based technique for unit testing. However, there still exist a vast array of specific techniques fulfilling these criteria such as *equivalence partitioning*, *boundary-value analysis* or *decision table testing* [11, 17]. Selecting one of these techniques and adapting it to our context of application is one of our major challenges.

- **What specific technique should be used to measure the adequacy of the tests?** We concluded that using a fault-/error-based test adequacy criterion is probably the best option for the context of our problem. However, once again we must still select one of the many available techniques to measure test adequacy using those criteria such as *error seeding*, *program mutation* or *perturbation testing* [17].

- **How should the test cases be specified?** The specification of the test cases should be rigorous and clear in order to provide all the information needed (e.g. inputs, expected outputs, etc.) in a widely accepted format. For that purpose, we plan to revise the literature of software testing and the related standards [1].

- **Which criteria should be followed to generate the input FMs?** The criteria for the generation of input data usually depend on the testing technique used. However, these techniques are often designed to work with numeric values and not with complex structures as FMs. Thus, the adaptation of these criteria to the context of FMs is a key challenge to decide the number, size and form of the input FMs to be used in our tests.

- **How could our proposal be integrated into the FAMA framework?** FAMA (FeAture Model Analyzer) is an extensible framework for the automated analysis of FMs integrating different logic paradigms and solvers. Our final goal is to integrate our proposal into this framework in order to enable the automated testing of the tools for the automated analysis of FMs.

References

[1] Draft IEEE Standard for software and system test documentation (Revision of IEEE 829-1998). Technical report, 2007.

[2] L. Baresi and M. Pezzè. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, 2006.

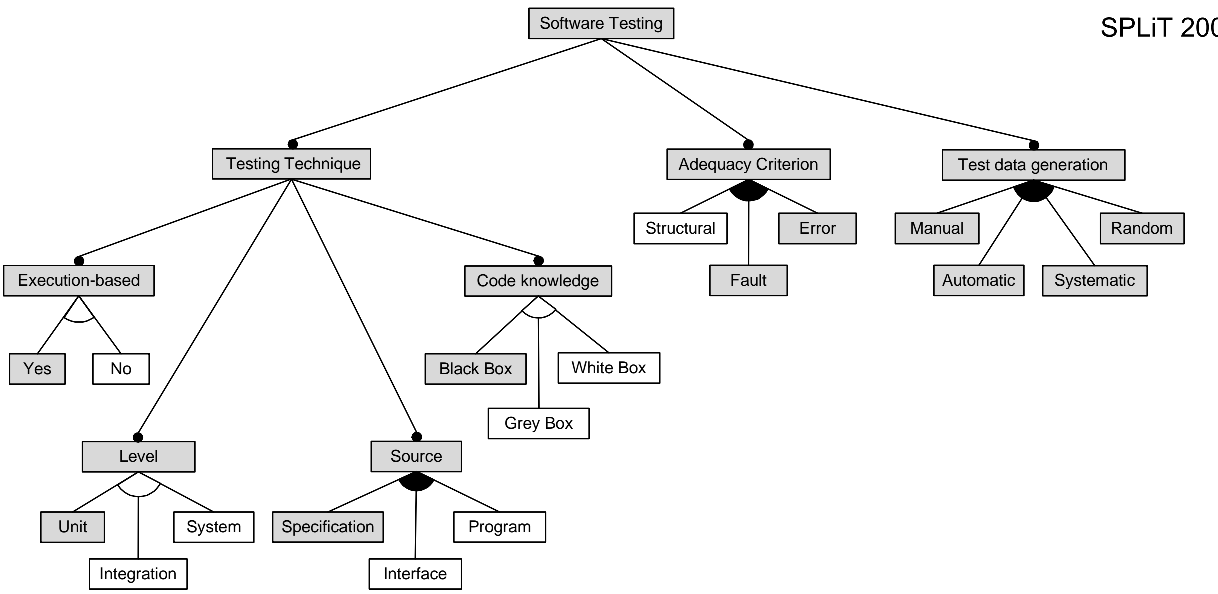


Figure 1: Feature-based overview of software testing methods

[3] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.

[4] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[5] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. *LNCS*, 4143:389–398, 2006.

[8] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] S. Fan and N. Zhang. Feature model based on description logics. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1144–1151. 2006.

[10] G. Kapfhammer. *The Computer Science Handbook*, chapter Software Testing. CRC Press, 2nd edition, June, 2004.

[11] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[12] S. Schach. Testing: principles and practice. *ACM Comput. Surv.*, 28(1):277–279, 1996.

[13] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software, in press*, 2007.

[14] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez. Fama framework. In *Proceedings of the 12th International Software Product Line Conference (Tool demonstration)*, 2008.

[15] H. Wang, Y.F. Li, J. un, H. Zhang, and J. Pan. Verifying Feature Models using OWL. *Journal of Web Semantics*, 5:117–129, June 2007.

[16] W. Zhang, H. Zhao, and H. Mei. A propositional logic-based method for verification of feature models. In J. Davies, editor, *ICFEM 2004*, volume 3308, pages 115–130. Springer–Verlag, 2004.

[17] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Model-based Test Design for Software Product Lines

Erika Mir Olimpiew
Computer Sciences Corporation
eolimpiew@csc.com

Hassan Gomaa
Department of Computer Science
George Mason University
hgomaa@gmu.edu

Abstract

A model-based testing method for software product lines (SPL) is used to create test specifications from use case and feature models, which can then be configured to test individual applications that are members of the SPL. This paper describes a feature-oriented model-based testing method for SPLs that can be used to reduce the number of test specifications created to cover all use case scenarios, all features, and selected feature combinations of a SPL. These test specifications can be automatically selected and configured during feature-based test derivation to test a given application derived from the SPL.

1. Introduction

Use-case based testing methods for SPLs extend use cases, or functional models developed from these use cases, to be configurable for an application derived from a SPL [1-4]. These methods design and customize reusable tests for a fixed set of applications derived from the SPL.

However, in some situations it is not possible to predict the set of applications that will be customized by all end users of an SPL. For example, a mobile phone SPL provides many optional features that can be selected by a mobile phone user.

In these situations, feature-based testing methods for SPLs can be applied to determine what application configurations to test [4, 5]. However, these methods do not address how to apply feature-based test derivation to customize reusable tests for the selected applications.

This paper describes how a feature-based testing method can be combined with a use case based testing method for SPLs, to reduce the number of reusable test specifications that need to be created for a set of applications selected from an SPL. Customizable Activity Diagrams, Decision Tables and Test Specifications (CADeT) is a functional testing method for a SPL that creates test specifications from both the use case and feature models of a SPL. A feature-based coverage criterion is applied together with a use case - based coverage criterion to cover all features, all use case scenarios, and selected feature combinations of a SPL. CADeT reduces the number of reusable test specifications that need to be created to cover all use case scenarios of the selected applications.

The rest of this paper is organized as follows. Section 2 describes the SPL development approach used in this research; section 3 gives an overview of CADeT. A more detailed description of CADeT is described in sections 4 (create reusable activity models), 5 (create decision tables) and 6 (configure reusable test models). Related research on model-based testing is described in section 7, and the conclusions are in section 8.

2. Software Product Lines

SPL development consists of SPL engineering and application engineering (Figure 1). Model-based SPL engineering consists of the development of requirements, analysis and design models for a family of systems that comprise the application domain. During application engineering, an application configuration is derived from the SPL, which includes all the common features and selected optional and alternative features. The requirements, analysis, and design models, as well as component implementations, are also customized based on the features selected for that application. Any unsatisfied requirements, errors and adaptations are addressed iteratively in SPL engineering.

Product Line UML-Based Software Engineering (PLUS) is a feature-based multiple-view modeling design method for software product lines. In the Requirements phase, PLUS uses feature modeling to model variability and use case modeling to describe the SPL functional requirements [6]. The relationship between features and use case is explicitly modeled by a feature/use case dependency table.

A *feature model* [7] distinguishes between the members of the SPL in terms of their commonality (expressed as common features) and variability (optional and alternative features), where a *feature* is the name of a requirement or characteristic of a SPL. A feature model also describes feature dependencies, and

feature groups that define constraints in combining features.

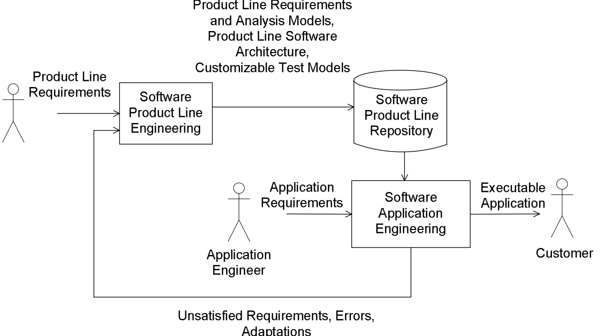


Figure 1 SPL development processes

3. Extending PLUS with CADeT

CADeT extends PLUS [6] to create functional models that can be used to generate functional system test specifications. Figure 2 shows how CADeT (shaded in gray) impacts the PLUS method (Gomaa 2005). A SPL engineer develops the SPL requirement models, analysis models, and software architecture using PLUS. Then, a test engineer uses CADeT to develop customizable activity diagrams, decision tables, and test specifications from the feature and use case requirements models. An application engineer applies feature-based application derivation to derive one or more applications from the SPL, while a test engineer uses CADeT to apply feature-based test derivation to select and customize the test specifications for these applications. Any unsatisfied requirements, errors and adaptations are sent back to the SPL engineer, who changes the reusable assets and stores them in the SPL repository.

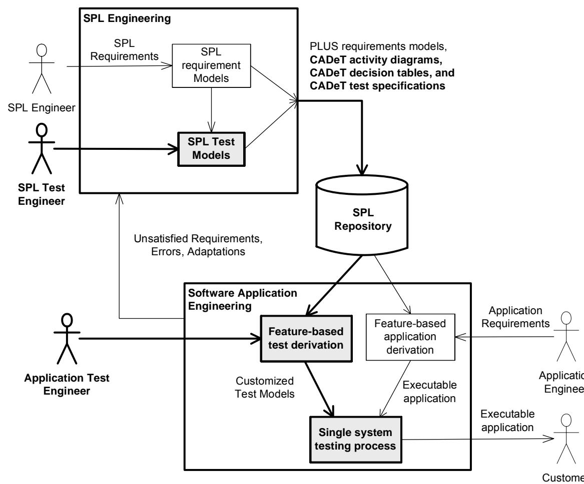


Figure 2 Extending PLUS with CADeT

The process of creating reusable activity models, decision tables and test specifications using CADeT is described next.

4. Reusable Activity Models

Functional models, such as activity diagrams, can be used to make the sequencing of activities in a use case description more precise for analysis and testing.

UML stereotypes [8] are used in CADeT to distinguish between different granularities of functional variability in the activity diagrams of a SPL, and feature conditions are used to relate this variability to features in the feature model.

An activity diagram is created from each use case description in the use case model, and then the activities in the activity diagrams are associated with features in the feature model. The following activity node role stereotypes are used to distinguish between different granularities of functional abstraction:

- A «use case» activity node, which describes a use case.
- An «extension use case» activity node, which describes an extension use case.
- An «included use case» activity node, which describes an included use case.
- An «aggregate step» activity node, which groups a sequence of activities, or events, in a use case description.
- An «input step», which describes an input event from the actor to the application in a use case description.

- An «output step», which describes an output event from the application to the actor in a use case description.
- An «internal step», which documents an internal (non-observable) activity in the application.

The impact of common, optional, and alternative features on the activity diagrams is analyzed with the feature to use case relationship table of PLUS [6]. A feature to relationship table associates a feature with one or more use cases or variation points, where a variation point identifies one or more locations of change in the use cases [6]. A reuse stereotype is a UML notation that classifies a modeling element in a SPL by its reuse properties [6]. The following reuse stereotypes describe how an activity node is reused in the applications derived from the SPL:

- A «kernel» activity node, which corresponds to a «common» feature in the feature model.
- An «optional» activity node, which corresponds to an «optional» feature in the feature model.
- A «variant» activity node, which corresponds to an «alternative» feature in the feature model.
- An «adaptable» activity node, which identifies an activity node that is associated with a use case variation point.

Besides reuse stereotypes, feature conditions are added to associate the variability in the control flow of an activity diagram with a feature in a feature model. A *feature condition* is a parameter variable, which contains mutually exclusive feature selections such as (true or false) values. Setting the value of a feature condition enables or disables the activities associated with the feature in the activity diagram of an application derived from the SPL.

4.1. Example of activity diagram

Table 1 shows the feature conditions and feature selections associated with features of the Banking System SPL. Figure 3 shows an excerpt from a use case model of the Banking System SPL, and Figure 4 shows a simplified activity diagram created from the Validate Pin use case description. The English, Spanish and French language features correspond to the vpLanguage variation point in the feature to use case relationship table, which impacts all output steps in the activity diagram of Figure 4. Each of these activity nodes is stereotyped as «adaptable».

Table 1 Feature conditions of the Banking System SPL

Feature condition	Feature selections
BankingSystemKernel	T
onlineBanking	{T, F}
greeting	{standard, enhanced}
language	{Eng, Spa, Fre}
pinFormat	[3..10]
maxAttempts	[1..5]

Figure 3 Excerpt of use case model for Banking System SPL

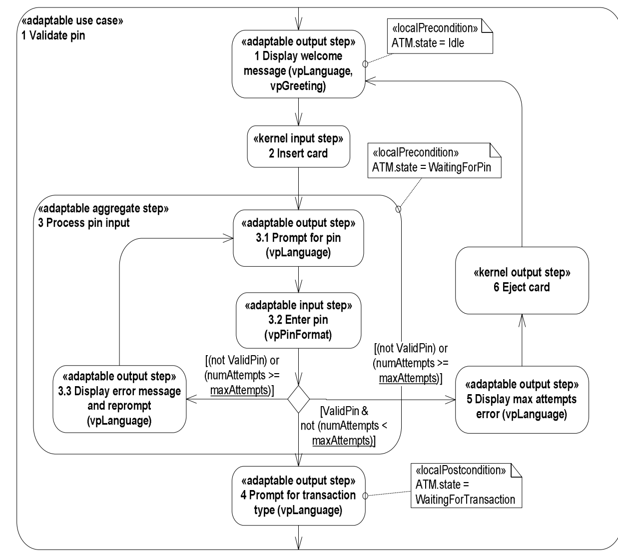


Figure 4 Activity diagram for Validate Pin use case

5. Decision Tables

Decision tables are used in CADeT to represent and organize the associations between features and test specifications created to cover the use case scenarios in a SPL. Decision tables are created from each activity diagram in the SPL. The precondition, feature conditions, execution conditions, postconditions, and activity nodes of the activity diagram are mapped to condition rows in the decision table. Simple paths are traced from an activity diagram for each use case scenario and then mapped to a reusable test specification in a column in the decision table. A *simple path* is a sequence of unique, non-repeating activity nodes traced from an activity diagram.

A feature can be associated with a test specification created for a use case scenario (a unit of coarse-grained functionality), or a feature can be associated with a

variation point in that test specification (a unit of fine-grained functionality). A variation point in a test specification in a decision table is represented using the «adaptable» stereotype. Fine-grained variations tend to be dispersed and repeated within the activity diagrams, so managing these variations requires more sophisticated techniques to group related functions and to minimize redundancy in the test models.

CADeT distinguishes between the binding times of coarse-grained functional variability (feature to test specification) and fine-grained variability (feature to variation point). The feature selections of features associated with test specifications are bound during SPL engineering, while the feature selections of features associated with variation points are bound during feature-based test derivation. Delaying the binding time of the fine-grained variability improves the reusability of the test specifications by reducing the number of test specifications that need to be created and maintained for a SPL.

5.1. Example of decision table

Table 2 shows a simplified decision table for the Validate Pin use case, which has four test specifications that correspond to paths traced from the activity diagram in Figure 4 for three scenarios. These test specifications contain vpLanguage, vpPinFormat, and vpPinAttempts variation points. The vpLanguage variation point occurs in every test specification, and is impacted by the alternative English, French, and Spanish language features. Thus, the feature selections of the language features {Eng, Fre, Spa} are entered in the intersection of each test specification with the language feature condition.

Table 2 Decision table for Validate Pin

1 Validate Pin	**«adaptable» Init sequence**	**«adaptable» Pin is valid**	**«adaptable» Pin is invalid**	**«adaptable» Pin is invalid max times**
Feature conditions: BankingSystemKernel	T	T	T	T
language	{Eng, Spa, Fre}	{Eng, Spa, Fre}	{Eng, Spa, Fre}	{Eng, Spa, Fre}
pinFormat		[3..10]	[3..10]	[3..10]
maxAttempts		[1..5]	[1..5]	[1..5]
Preconditions: ATM	Idle	WaitingForPin	WaitingForPin	WaitingForPin
Execution conditions: ValidPin	-	T	F	F
numAttempts >= maxAttempts	-	F	F	T
1 «adaptable output step» Display welcome msg (vpLanguage)	√			
2 «kernel input step» Insert card	√			
3.1 «adaptable output step» Prompt for pin (vpLanguage)		√	√	√
3.2 «adaptable input step» Enter pin (vpPinFormat)		√	√	√
3.3 «adaptable output step» Display error message and re-prompt for pin (vpLang)			√	
4 «adaptable output step» Prompt for transaction type (vpLanguage)		√		
5 «adaptable output step» Display max attempts error (vpLanguage)				√
2 «kernel output step» Eject card				√
Postconditions: ATM	WaitingForPin	WaitingFor Transaction	WaitingForPin	Idle

5.2 A Variability Mechanism to Customize Decision Tables

A variability mechanism is a technique that enables the automatic configuration of variability in an application's requirements, models, implementation and test specifications. CADeT contains a tool suite that uses a parameterization variability mechanism to customize the decision tables, and then generate test specifications from these tables for an application derived from the SPL. In order to automate the customization of the fine-grained variability, the decision tables need to be modified to describe the

variations associated with each variation point, and these variations need to be linked to features in a feature list, which describes all feature conditions and possible feature selections of the SPL. Selecting the value of a feature condition selects the test specifications associated with that feature, and also customizes the fine-grained variability in the test specifications.

6. Configuring Reusable Test Models

A feature-based testing method for SPLs as in [4, 6] can be applied to determine what applications to test. Then, CADeT can be used to customize the test specifications for each of these applications during feature-based test derivation. CADeT has tools to automate feature-based selection and customization of test specifications for each application.

First, the feature selections in the feature list of the SPL are set to correspond to the feature selections of an application derived from the SPL. This customizes the decision tables for that application. Next, a test specification generator tool is used to generate a test specifications document from the customized decision tables for the application.

Then, a test procedure definition tool is used to create a test procedure document for the application. The test procedure document describes a collection of system tests, where a system test describes the order in which a sequence of test cases (test specification instances) will be executed for an application derived from the SPL. A test order graph is constructed from the customized decision tables during feature-based test derivation. The test specifications are sorted by pre and post conditions to constrain the order in which these specifications can be executed for the application. Then, a test engineer traces paths through the test order graph to create system tests for the application.

7. Related research on model-based testing

Model-based testing creates test specifications from formal or semi-formal software models of a software application [9-13]. Combinatorial testing techniques [14] select inputs that cover pair-wise, triple or n-way combinations of an application's parameters. Combinatorial test generation tools such as Jenny [15] generate inputs to satisfy a selected combinatorial test coverage criterion.

Several use case-based testing methods [1-4, 16] have been developed to address the problem of systematically reusing the test specifications of a SPL. Reuys et al [1] developed the ScenTED technique (Scenario-based TEst case Derivation), which expands the work in [2]. Test specifications are traced from the activity diagrams of a SPL to satisfy the branch coverage testing criterion, and then customized for an application derived from the SPL.

Other software models, such as a decision tree, have also been used to apply systematic reuse to the test assets of a SPL. Geppert et al [17] re-engineer a legacy system test suite and then use a decision tree to configure this test suite for an application of the SPL.

McGregor [4] first introduced the problem of selecting application configurations to test from the potentially large configuration space of an SPL. Scheidemann [5] describes a method of selecting a minimal set of application configurations, such that successful verification of this set implies the correctness of the entire SPL.

8. Conclusions

This paper has described CADeT, a new model-based testing method for SPLs that combines feature-oriented and use case-based functional testing approaches for SPLs. Reusable test specifications are created from SPL models, and then customized during feature-based test derivation for a set of representative application configurations. The feasibility of method was evaluated on a Banking System and Automated Toll System SPLs [18]. The results of each case study showed that the method was feasible, and could be used to substantially reduce the number of test specifications created for a SPL.

Future research will investigate how different variability mechanisms can be integrated with this approach, such as separation of concerns, and how this approach can be extended to address integration testing of SPL architectures.

9. Acknowledgements

Many thanks to Diana Webber who developed the initial version of the Banking System SPL, and to V. Vonteru, who created a SPL implementation. Thanks are also due to D. Anderson, L. Beauvais, H. Hussein, J. Pepper, and R. Rabbi for evaluating the feasibility of this method on the Banking System SPL.

8. References

1. Reuys, A., et al., *Model-Based Testing of Software Product Families.* Lecture Notes in Computer Science, 2005. **3520**: p. 519-534.

2. Kamsties, E., et al. *Testing Variabilities in Use Case Models*. in *Software Product-Family Engineering: 5th Int'l Workshop*. 2003. Siena, Italy.
3. Bertolino, A. and S. Gnesi. *PLUTO: A Test Methodology for Product Families*. in *Software Product-Family Engineering: 5th Int'l Workshop*. 2003. Siena, Italy.
4. McGregor, J.D., *Testing a Software Product Line*. 2001, SEI.
5. Scheidemann, K. *Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems*. in *10th Int'l Software Product Line Conference*. 2006. Baltimore, MD: IEEE Computer Society Press.
6. Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. The Addison-Wesley Object Technology Series. 2005: Addison-Wesley.
7. Kang, K., *Feature Oriented Domain Analysis*. 1990, Software Engineering Institute: Pittsburg, PA.
8. OMG, *Unified Modeling Language: Superstructure, version 2.1*. 2007, Object Management Group.
9. Mayrhauser, A.v., R. T. Mraz and P. Ocken. *On Domain Models for System Testing*. in *Proceedings of the 4th Int'l Conference on Software Reuse*. 1996. Orlando, FL.
10. Offutt, J. and A. Abdurazik. *Generating Tests from UML Specifications*. in *Proceedings of the 2nd Int'l Conference on the Unified Modeling Language*. 1999. Fort Collins, CO.
11. Poston, R., ed. *Automating Specification-Based Software Testing*. 1996, IEEE Computer Society Press: Los Alamitos, California.
12. Briand, L.C. and Y. Labiche. *A UML-Based Approach to System Testing*. in *Proc. 4th Int'l Conference on the Unified Modeling Language (UML)*. 2001. Toronto (Ontario, Canada).
13. Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*. 2002, Reading, MA: Addison-Wesley.
14. Cohen, D.M., et al., *The AETG System: An Approach To Testing Based on Combinatorial Design*. IEEE Transactions on Software Engineering, 1997. **23**(7): p. 437-444.
15. Jenkins, B., *Jenny: A Pairwise Testing Tool at http://www.burtleburtle.net/bob/math/jenny.html*. 2005.
16. Nebut, C., et al. *A Requirement-Based Approach to Test Product Families*. in *Software Product-Family Engineering: 5th Int'l Workshop*. 2003. Siena, Italy.
17. Geppert, B., J. Li, F. Roessler, and D. M. Weiss. *Towards Generating Acceptance Tests for Product Lines*. in *8th Int'l Conf. on Software Reuse*. 2004. Madrid, Spain: Springer-Verlag.
18. Olimpiew, E., *Model-Based Testing for Software Product Lines*. 2008, George Mason University: Fairfax, VA.

A Test-Driven Approach to Establishing & Managing Agile Product Lines

Yaser Ghanam, Shelly Park, Frank Maurer
University of Calgary
{yghanam, parksh, maurer}@cpsc.ucalgary.ca

Abstract

Test Driven Development (TDD) is an agile method that emphasizes writing tests before writing code as a means of 1) assuring the satisfaction of customer requirements, and 2) reinforcing good design habits. While the first objective is usually accomplished by acceptance tests, the second objective is achieved by unit tests. The tests also serve as a multilevel cohesive reference of the system specifications. We propose the use of this referencing mechanism – test artifacts – to establish and manage agile product lines. In this paper, we delve into some of the issues that need to be tackled before test artifacts are relied on as a driving force for reuse in product lines. These issues include establishing a framework for reuse, tests comparability, test traceability, test refactoring and test versioning. We also discuss the suitability of acceptance tests and unit tests as reusable artifacts, and we present a preliminary study to analyze their utilization.

1. Introduction

1.1. Background

As defined by Clement et al. [1], a product line is a group of software systems that share a common set of features. Because of this commonality, a high level of reuse occurs in software product lines allowing for a shorter delivery time and a considerable improvement in quality. What is also appealing about developing software families when compared to developing single systems is the opportunity for mass customization at a relatively low cost To efficiently handle the commonalities and the variations in the customer specifications, an intensive domain analysis takes place upfront to determine the scope of the product line, the points of variations, and the expected variants. Once the domain has been well understood, a flexible architecture (along with potentially reusable assets) is designed that is capable of accommodating potential members of the family and proactively address their variability. Then an application engineering process takes place to start instantiating members of the software family. This strict "domain-then-application" model, although has been deemed successful [2], is resource intensive and risky. It can be financially and managerially exhausting to small-to-medium scale organizations. Therefore, efforts to make software product lines more agile are increasingly important as the hope is that less upfront effort will be required and risk will be reduced by utilizing agile approaches in the development of product lines. An agile model to establishing and managing product lines may seem to be in conflict with the traditional practiced approaches, but resolving this conflict has a strong potential to cut down the upfront investment in domain analysis and architectural design. Blending agile methods and software product line engineering also has the advantage of introducing a minimalistic design philosophy that reduces or eliminates unnecessary investment in requirements that are too hard to predict in the long run. Finding which agile methods work best to address the different issues of product line engineering is a broad research area that has been gaining special attention in the past few years.

1.2. Motivation

TDD [3] is amongst the most commonly practiced agile methods. Contrary to traditional software development models that delegate testing to a late stage of the project life cycle, agile software development adopts a "test-first" philosophy that uses tests to aid with software design. That is, before implementing any feature, tests are to be written to determine what defines the completion of a feature. Once the tests are written, they are expected to fail because no implementation exists to satisfy them. As the development goes on, test cases start to gradually pass. Development stops as soon as all tests related to a given feature are passing and the customer representatives confirm that the feature is done. Generally, tests occur at two different levels. At a high level, where the customer is concerned, acceptance tests (AT) are produced as a well-defined, concrete agreement on what features are to be delivered. Acceptance tests are black box tests at the feature level and, hence, need not worry about the details of the implementation. At a lower level is unit testing (UT). Developers write unit tests as a means of ensuring the quality, modularity and neatness of the

design, and therefore they are highly dependent on the specifics of the implementation. Unit tests can occur at different granularities such as modules, classes and methods.

In agile contexts, ATs are thought of as a way of keeping a cohesive record of the specifications of the system under development. Since they are usually written in a format accessible to both technical and non-technical audiences, they can always be a reference of what was requested by the customer, what has been done so far and what is to be done next. Furthermore, with the assistance of tools like Fit [4], ATs can be automatically executed providing what is now known as "executable specifications".

This dependency on tests as a means of communicating the system requirements at different levels brings the idea that tests might be a good driving force for reuse in an agile software product line. That is, ATs and UTs can be the reusable artifacts that steer reuse of other artifacts in an agile product line. However, in order for tests to qualify for this responsibility, a raft of issues is to be considered such as tests reusability, comparability, traceability, refactoring, and versioning. *This paper discusses the different aspects of the proposed test-driven approach to shed light on what is to be done before we can establish a test-based agile product line*.

The remaining of this paper is structured as follows. The next section is a literature review of related efforts. Section 3 talks about the different issues that need to be addressed in order to establish test-driven agile product lines. Section 4 is a discussion about identifying features in product lines, furnished with an initial study of existing systems developed using a test-driven approach. Finally, we briefly go over some of our future plans to investigate the matter further.

2. Related Work

Unit testing is a mature field that has been discussed well in literature and successfully practiced in industry. In comparison, executable acceptance testing is a new agile technique that, supported by tools like FitClipse [5], is becoming increasingly popular. Melnik et al. [6] and Ricca et al. [7] studied the effectiveness of Fit acceptance tests in conveying requirement specifications and agreed that these tests do improve requirement understanding. Moreover, Park et al. [8] studied the benefits of executable acceptance tests. They assert that the tie between these tests and the source code allows for better progress tracking, enhances communication amongst stakeholders, and consequently leads to better software quality.

Integrating agile methods and SPLE practices is a new research area that is starting to gain attention in academia and industry. A handful of efforts proposed different ways to blend the two software paradigms. For example, Hanssen et al. [9] suggested the use of SPLE at the strategic level and the use of agile methods at the tactical level. Moreover, Paige et al. [10] proposed the use of Feature Driven Development (FDD) as a way to build product lines under special considerations for the product line architectural and component design.

3. Open Questions

Traditionally, domain analysis is the most crucial step in establishing software product lines. Not only is it a prerequisite to the application engineering step that is completed before applications are being developed, but is also what defines commonality, variability and reuse in the product line. To suggest an agile approach to build product families is to take the responsibility of defining an alternative model that satisfies the requirements of a successful product line practice. Our proposal is to use acceptance or/and unit tests to support this alternative. In this section, we discuss *some* of the major issues that need to be addressed in order for tests to qualify for this task.

3.1. A framework for reuse

For a test to be capable of driving reuse in the product line, it needs to be reusable *itself*. Tests that have previously been produced to drive development in applications of the same domain are to be refactored and reused whenever applicable. However, determining this applicability is not straightforward. We need first to put required features in the new system within the context of previously developed ones (the family) to search for a match from existing assets. This implies that we also need to know how to *compare* these assets. The other implications of wanting to establish a framework for reuse in an agile product line is to have a record of every instance of reuse for future maintenance, extension and quality improvement. This leads us to ask many questions about the *traceability* of reusable assets. One more issue that needs to be delved into is how to do the *refactoring* when a match is found. This topic is particularly significant considering the risks that arise from incorporating changes that might regressively have influence on a number of products in the family at the same time. Also, performing the refactoring process several times will result in different versions of the test artifact existing at different points of time. So how do we mange and keep track

of these *versions*, and ensure their consistency with the source code? The following subsections will talk about these issues in more detail.

3.2. Tests comparability

In the collaborative activity of writing acceptance tests, the customer and the developers produce tests that describe the features in the system under development. Later, when a new system (in the same domain) is being developed, a new set of acceptance tests are written for the same purpose. In order to crunch knowledge about whether the newly requested features correspond to the previously developed ones, there needs to be a mechanism to compare tests to realize how similar they are. This is especially tricky because we need first to clearly define what it means for two tests to be similar, what metrics should be used to determine similarity, and what degree of similarity is satisfactory to make an economical reuse. We may need to rely on mathematical and economical models to answer these questions and make it feasible to automate the comparison process.

3.3. Test traceability

It is clear that reuse without traceability is impractical. For an asset to be reused across different products, there has to be a mechanism to trace this reuse so that future reference is feasible. Say after reusing a specific module in a number of different applications, a bug is reported and needs to be fixed. With the unavailability of comprehensive documents stating where reuse has occurred in the product line, it would be very time consuming to go over all the products in the family and look deep at the module level to figure out which members of the family has used this specific module. The other alternative, and a more practically appealing one, is to employ tests to trace code reuse across different products. That is, a database stores information about all reuse instances that have occurred in the product line. How sophisticated this database needs to be is a function of the intricacy of the domain, and how much the product line has matured in the evolutionary process. Furthermore, tracing tests in a product line provides the advantage of being able to run regression tests across all products in the family when any of the reusable features experiences modification to make sure the modification has no side effects.

3.4. Test refactoring

The vision of the stakeholders becomes clearer as the project evolves over time resulting in change requests being initiated. Assuming a healthy practice of a test-driven approach, development occurs only with accordance to written tests. Therefore, when a change is to occur in the source code, all corresponding tests have to be updated. The process of updating these tests is time-consuming and can be risky, especially for acceptance tests. Because, according to Ordelt et al. [11], acceptance tests "lack the regression safety net that production code has." This issue becomes more incisive when considering a family of systems where change is more likely to occur and affect a wider range of source code. That is, as the product line architecture evolves over time, the separation between the commonality layer and the variability layer of reusable assets ossifies as a result of the continuous refactoring of core assets. This refactoring process needs to be reflected in associated tests to ensure consistency. Ordelt et al. [11] has studied this matter, and extended FitClipse [5] to support acceptance test refactoring using the Eclipse refactoring framework.

3.5. Test versioning

Continuous integration is a vital practice in agile software development. Developers continuously commit the changes they make to a repository and wait for a confirmation that the project has incorporated their changes successfully. If anything goes wrong in the new build, it is always possible to go back to an older error-free state. The rule of thumb is that developers should never check out broken code. Being able to go back and forth between different versions of the system is a necessity not a luxury for a successful continues integration practice. Unit tests are usually imbedded within the same project containers, and therefore their version need not be addressed separately (they already are versioned together with the code). This is not the case with acceptance tests, however. For they are built and executed separately in tools like Fit, associating versions to them is not common. This might not be an issue when thinking about single systems where a simple one-to-many relationship exists between the system and its acceptance tests. However, the issue becomes a quagmire in product families due to the complex many-to-many relationships that occur between applications and reusable assets.

An interesting approach would be to develop a framework that incorporates the different versions of test artifacts within the versioning system in the repository. A mechanism to support traceability of artifacts in these repositories, as discussed earlier, can also be integrated to make the versioning process of test artifacts more meaningful in the product line context.

4. Identifying Features in Product Lines

Tests hold some important information about the design and quality requirements and allow the developers to extract the common code based on functional features. Therefore, we want to find out if tests can be used to determine the commonalities in software systems of the same domain. In test-driven development, there are two types of tests. Executable acceptance tests are specified by the customer and they communicate the system requirements in testable format. These tests are defined in terms of functional features. Unit tests are written by the developers and they communicate the structural quality requirements. Unit tests are defined in terms of structural features. By writing the tests first, the tests become essential artifacts that communicate functional and structural requirements of the software. Instead of heavy up-front design of the software product line, which can be costly, we want to investigate whether these tests can support the establishment of an agile software product line. Our research question is to find out which type of tests provides richer information about the code.

Let's suppose that these tests carry very important information about the product line and that we can use these tests to extract the code for the common layer easily. As part of the research, we decided to measure the code coverage using two different types of tests in order to determine the traceability of the tests to the code. Our assumption is that Fit tests provide functional features, but may not necessarily trace well to the code as acceptance tests are based on black-box testing concept. Unit tests provide better traceability of test to code, but it may be too fine grained. Additionally, we are interested in finding out the intersecting code coverage by these two types of tests in order to determine the traceability of code from acceptance tests to unit tests code and ultimately to the code. If this relationship does exist, then upgrading one Fit test to the common layer would tell the developers which part of the code or unit tests must also be upgraded to the common product line layer.

Although code traceability or code coverage is irrelevant for acceptance tests, all functional features that are defined in the acceptance tests should be traceable to some parts of the code. We want to find out how much of the code can be traceable via acceptance tests in order to easily identify the code that needs to be upgraded to the common layer of the software product line.

4.1. Study Setup

We have taken two software projects and analyzed the code coverage. The two software products used for the experiment are FitClipse [5] and Flickr Desktop [12]. FitClipse is built for academic research purpose. It is a support tool for Fit tests and runs as an Eclipse plug-in. FitClipse is developed by numerous graduate and undergraduate students over the years. It has about 5500 lines of code. Flickr Desktop allows the user to browse pictures on Flickr website [13] and displays the selected pictures as desktop background in a slide show format. The Flickr Desktop project was a product of a course work by about a dozen students. It has about 1500 lines of code. Both software products used Fit tests for acceptance tests. We decided to measure the code coverage based on statement coverage criteria and measured the intersecting code coverage by both Fit tests and unit tests. Intersecting code coverage refers the lines of code that were executed by both Fit tests and Unit tests. The code coverage was measured manually due to the lack of code coverage tool for Fit tests.

4.2. Data Analysis

Table 1 summarizes the results obtained from the analysis. FitClipse have 20 Fit tests and 119 unit tests. Fit tests had 13% code coverage and unit tests had 14% code coverage. Only 46 lines of code were actually used by both Fit tests and unit tests. Flicker Desktop had 26 Fit tests and 59 unit tests. Fit tests and unit tests both covered 48% of the code. However, only 36% of the code was executed by both Fit tests and unit tests.

Table 1: Code Coverage Result for Fit and Unit Tests

	Fit	Unit
FitClipse		
Number of Tests	20	119
Code Coverage	692/5469 (13%)	747/5469 (14%)
Intersecting Code Coverage	46/5469 (0.8%)	
Flickr Desktop		
Number of Tests	26	59
Code Coverage	725/1509 (48%)	730/1509 (48%)
Intersecting Code Coverage	549/1509 (36%)	

The analysis shows an interesting initial result. Fit tests can cover comparable amount of code compared to unit tests with fewer number of tests. For FitClipse,

20 Fit tests were able to achieve similar code coverage as that of 119 unit tests. Likewise, 26 Fit tests were able to achieve the same code coverage as 59 unit tests in Flickr Desktop. Acceptance tests define a feature when unit tests define much more refined structural testing specifications. The reason for more code coverage by Fit tests is due to the setup that is needed in order to execute the functional specification. Unit tests often did not require extensive data or code setup to execute.

The two software products used for the analysis have relatively low test coverage. Due to the lack of time and money, achieving full test coverage is often very difficult. Given these constraint, it is interesting how Fit tests and unit tests do not necessarily test for the same things or even similar part of the code when the coverage is limited. For example, FitClipse has only 0.8% of the code that is executed by both Fit tests and unit tests. In Flickr Desktop, only 36% of the code is tested by both types of tests. The result suggests that the developers and customers do not necessarily think alike when it comes to testing and they view importance of the testing features differently given limited time to write the complete set of tests.

Given that most of industrial software development projects also do not achieve full test coverage, the early experiment results points that we need to look at both acceptance tests and unit tests for the purpose of product line engineering. Unit tests may be too fine grained, as seen in the results for FitClipse that 119 unit tests had similar code coverage as 20 Fit tests. Unit tests seem to test too fine grained structure of the code for extracting common layer code. Fit tests seem to provide higher level of system abstraction information, because each test mapped to larger set of code. However, it is often not possible to trace the fit tests to the corresponding unit tests.

The early result provides an interesting look into the relationship between acceptance tests and unit tests when the software project does not have enough time to achieve full test coverage. However, is there a point when having more Fit tests does not necessarily increase the code coverage? Although the main purpose of acceptance tests is not about achieving full code coverage, as the concept of code coverage is from structural testing practice, a functional feature from acceptance tests should be traceable to some parts of the code. In addition, it would be also interesting to look at the reasons for the intersecting lines of code coverage. We need to find out the contributing causes that influence why unit tests and fit tests do not necessarily test for the same parts of the code. Do developers and customer really think differently? Or do they have different testing priorities? Although code coverage is not the aim of acceptance tests, the result shows an interesting phenomenon that developers and customer do not necessarily test the same feature. The results show that they ended up testing the different parts of the code. Even looking at the result from redundancy point of view, these tests serve different purposes and thus redundancy does not mean waste of effort: acceptance test is for supporting customer collaboration and unit tests are for improving software design. The test coverage is simply a side effect. However, it would be interesting to know why the developers did not write unit tests for the portions that were covered by the Fit tests as the developers clearly did encounter these code while writing the fixtures code that attached the Fit test specifications to the code. The preliminary result proposes the interesting question of whether the developers and customers view the system differently and thus give different emphasis on the features that require testing. We need to take a look at more industrial projects to see if the phenomenon persists.

6. Conclusions

Developing software families is a successful paradigm that promises considerable cuts in cost and development efforts accompanied with enhancements in quality. Introducing agile software development practices to software product line engineering aims at magnifying these advantages even further. In this paper, we proposed the use of test artifacts to establish and manage agile product lines. We highlighted some open questions that need to be researched before tests can be utilized as a driving artifact for reuse. A preliminary analysis of two systems was also discussed to shed light on the roles that acceptance tests and unit tests could play in the context of a test-driven product line. The initial results suggest that because developers and customers view the system differently, they give different emphasis on what is to be tested. Therefore, more is to be done to understand the multimodality of test artifacts and their appropriateness to steer reuse. Currently, we are preparing a controlled experiment to better understand the factors that contribute to how stakeholders write tests, and how developers translate these tests into design decisions. We also aim to conduct a series of experiments to investigate issues related to test-driven reusability and refactoring.

7. References

[1] Clements, P., and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[2] Pohl, K., Böckle, G., and Linden, F., *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, Germany, 2005.

[3] Beck, K., *Test Driven Development- by Example*. Boston: Addison-Wesley, 2003.

[4] FIT: Framework for Integrated Test, http://fit.c2.com, last accessed on May 29th, 2008.

[5] FitClipse, http://ase.cpsc.ucalgary.ca/index.php/FitClipse, last accessed on May 29th, 2008.

[6] Melnik, G., Read, K., and Maurer, F., Suitability of fit user acceptance tests for specifying functional requirements: Developer perspective. In *Extreme programming and agile methods - XP/Agile Universe 2004*, pp. 60–72.

[7] Ricca, F., Torchiano, M., Ceccato, M., and Tonella, P., Talking tests: an empirical assessment of the role of fit acceptance tests in clarifying requirements. *The 9th International Workshop on Principles of Software Evolution 2007, Croatia.*

[8] Park, S., and Maurer, F., The benefits and challenges of executable acceptance testing. *The APSO Workshop - collocated with the 30th International Conference on Software Engineering 2008,Germany.*

[9] Hanssen, G., and Fægri, T., "Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering", accepted for special Issue of *Journal of Systems and Software (JSS)*, 2008.

[10] Paige, R., Xiaochen, W., Stephenson, Z., and Phillip J., “Towards an Agile Process for Building Software Product Lines”, *LNCS: XP 2006*, pp. 198 – 199.

[11] Ordelt, H., and Maurer, F., Acceptance test refactoring. *Proceedings of the 9th International Conference on Agile Processes and eXtreme Programming*, Springer, Ireland, 2008.

[12] Flickr Desktop, http://sourceforge.net/projects/flickr-desktop, last accessed on May 20th, 2008.

[13] Flickr, www.flickr.com, last accessed on May 30th, 2008.

Formalizing platform-independent Test Cases for Model-driven Software Development

Eugen Fischer[1], Peter Knauber[2]
[1]InterComponentWare AG, Walldorf (Baden), eugen.fischer@icw.de
[2]Mannheim University of Applied Sciences, p.knauber@hs-mannheim.de

Abstract

Model-driven software development (MDSD) can be seen as one specific implementation of software product line engineering. Accordingly, similar effects arise wrt. testing of product line instances: product line engineering is capable of dramatically increasing the product development efficiencies, and thus can outpace if not overwhelm traditional test practices[1]. This paper describes an approach to formalize test cases in a way that they are reusable over the different product line instances and its application in industrial practice.

1. Motivation

Model-driven software development (MDSD) and model-driven architecture (MDA) as described by the OMG can improve software development due to several factors. One of the main arguments for MDA is the separation of "business and application logic from the underlying platform technology" [2]: one high-level model that can describe a solution in a kind of domain-specific language can be transformed into code for different platforms (e.g., J2EE, .NET) just by using different generators.

But exactly this fast and easy development approach gives testers a hard time because they typically have to manually(!) code automated test cases for each target platform in order to ensure that the software is working properly on each of these.

The alternatives to avoid this repeated and error-prone effort are the representation of test cases in a platform-independent format plus either their interpretation on the target platforms or the generation of executable code for these platforms.

2. The Approach

At InterComponentWare AG (ICW), Walldorf, a development kit for the integration of customer-specific systems with ICW's eHealth Telematics Infrastructure is developed for (currently) three platforms (J2EE/Java, .NET/C#, and COM/VB). In this context, both alternatives mentioned in section 1 were researched in detail during a diploma thesis [3] and prototypes were implemented. The interpretation (using JavaScript as test case representation and respective interpreters) was discarded due to several limitations of the approach.

Instead, the generation approach was taken: a (prototype) tool enables the comfortable definition of test cases based on information of the same UML model which is used for code generation. The test cases are encoded in XML format which in turn is used for the generation of xUnit test code that can be manually extended (if necessary) and run on the target platforms.

3. Conclusion

The approach described fits seamlessly with the model-driven development approach used at ICW. Initial case studies show that the effort for test case development for the three platforms can be reduced by up to 40 percent using the approach and the tool described. Currently it is planned to apply them within a larger upcoming project.

References

[1] Peter Knauber, William Hetrick: "Product Line Testing and Product Line Development—Variations on a Common Theme", ALR-2005-0xx: 2nd International Workshop on Software Product Line Testing (SPLiT 2005), September 2005, Rennes, France

[2] Object Management Group (OMG): "Model Driven Architecture (MDA)", www.omg.org/mda

[3] Eugen Fischer: "Formalisierung von plattformunabhängigen Testfällen zur Generierung von Modultests im Umfeld modellgetriebener Softwareentwicklung", Diploma Thesis, Mannheim University of Applied Sciences, 2008 (in German)

How SOA Testing Could Benefit from Software Product Line Testing*

Andreas Metzger
SSE, University of Duisburg-Essen, Schützenbahn 70, 45117 Essen
andreas.metzger@sse.uni-due.de

1 Motivation and Introduction

The Service Oriented Architecture (SOA) is increasingly adopted by industry as a paradigm for building highly dynamic, distributed and adaptive software applications. The SOA shares some similarities with software product line (SPL) engineering; e.g., both paradigms promote high levels of reuse to build flexible and cost-effective software applications. To a certain extent they also share similar challenges for what concerns testing. Below, we briefly describe those challenges and how they could be addressed.

The SOA provides several key principles for building service-based applications (SBAs) [1, 2]. Thanks to these principles, SBAs meet the requirements for dynamism and adaptation. A SBA is realized by "composing" software services. For the service composer, a software service is not an individual piece of software. Rather, it represents some functionality that can be invoked through the service's interface, where the actual software that implements this functionality is executed, maintained and owned by the provider of that service. Ideally, the service composer can select from a vast amount of readily available services from a diverse range of service providers.

2 Challenges in SOA Testing

One key principle of the SOA is loose coupling, which means that a service within a service composition only makes weak assumptions about its interactions with other services in the composition; e.g., instead of services being tightly coupled by means of a common data model, only simple data types are used. This principle allows a service to be (re-)used in many different service compositions. Obviously, this form of reuse requires considerable testing effort. In principle, one would need to test each service in all potential compositions. Due to the unbound number of potential service compositions, a comprehensive test thus is not possible. Therefore, an interesting challenge in SOA testing is how to address this complexity issue.

Another key principle of the SOA is late binding. This principle implies that many services will be discovered and composed into a SBA only at run-time. As a consequence, not all services that will be part of the actual SBA will be known during design time. Thus, one challenge is how to handle those missing parts during testing.

3 Similarities to SPL Testing

The above challenges resemble issues that one also faces in SPL testing. As an example, SPL testing needs to cope with a similar complexity problem like SOA testing. Industry reports on product lines that range to up to tens of thousands of variation points and variants, leading to a very huge number of potential product line applications. Consequently, SPL research has addressed how to handle this huge number of potential applications during quality assurance (e.g., see [3]).

Further, when testing the reusable artifacts of a SPL during domain engineering, typically not all variants have been implemented. Thus, like in the SOA case, one needs to handle those missing parts; e.g., by creating placeholders [4].

As these two examples show, similarities between SOA and SPL testing exist. We thus believe that joint research between the SPL and the SOA communities can produce relevant solutions that are applicable in both areas.

References

[1] T. Erl. *Service-oriented Architecture*. Prentice Hall, New York, NY, 2004.
[2] N. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.
[3] A. Metzger. Quality issues in software product lines: Feature interactions and beyond. In *Feature Interactions in Softw. and Comm. Systems IX*, pages 1–12. IOS Press, 2007.
[4] S. Reis, A. Metzger, and K. Pohl. Integration testing in software product line engineering: A model-based technique. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 321–335. Springer, 2007.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube). For further information, please visit http://www.s-cube-network.eu/

Functional Testing of Feature Model Analysis Tools. A First Step *

Sergio Segura, David Benavides and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
University of Seville
Av Reina Mercedes S/N, 41012 Seville, Spain
{sergiosegura, benavides, aruiz} AT us.es

1. Extended Abstract

The analysis of a Feature Model (FM) consists on the observation of its properties. Typical operations of analysis allow finding out whether a FM is void (i.e. it represents no products), whether it contains errors (e.g. feature that can not be part of any products) or what is the number of products of the software product line represented by the model. In particular, the automated analysis of FMs is generally performed in two steps: $i)$ First, the model is translated into a specific logic representation (e.g. Constraint Satisfaction Problem (CSP), $ii)$ Then, off-the-shelf solvers are used to automatically perform a set of analysis operations on the logic representation of the model [1].

Implementing the operations for the analysis of FMs using a specific solver is not a trivial task. The lack of specific testing mechanisms in this context difficult the development of tools and reduce their reliability. To improve this situation, we propose to design a set of generic test cases to verify the functionality and correctness of the tools for the automated analysis of FMs. These test cases would help to improve the reliability and robustness of the existing tools while reducing the time needed to develop new ones. As a starting point, in this position paper we narrow the search for a suitable testing technique, test adequacy criteria and test data generation mechanism to be used for our tests. For that purpose, we overview some of the classification of software testing methods reported in the literature [2, 3] and study the adequacy of each approach to the context of our problem. In particular, we address the following classification criteria:

- **Testing technique.** For the selection of a suitable testing technique we try to classify our testing approach according to: $i)$ Expected knowledge of the source code (i.e. black-box, white-box or grey-box testing), $ii)$ Source of information used to create the tests (i.e. program-based, specification-based or interface-based testing), $iii)$ Testing time (i.e. Execution-based *vs* non-execution-based testing), and $iv)$ Testing level (i.e. unit, integration or system testing).
- **Test adequacy criteria.** In order to define the quality of our tests we evaluate three generic adequacy criteria: fault-based, error-based and program-based criteria.
- **Test data generation.** For the selection of a suitable test data generation mechanism we mainly consider two choices: $i)$ Manual *vs* automated, and $ii)$ Random *vs* systematic.

As a result of the evaluation of these alternatives, we conclude that we should use a black box, execution- and specification-based technique for unit testing. We also consider that both, fault-based and error-based adequacy criteria are a suitable option to validate our test cases. Finally, we devise that all the alternatives studied for test data generation could be useful in our future test suite.

As a final remark, we state some of the research questions to be addressed in our future work.

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533)

References

[1] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[2] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Model-based Test Design for Software Product Lines

Erika Mir Olimpiew
Computer Sciences Corporation
eolimpiew@csc.com

Hassan Gomaa
Department of Computer Science
George Mason University
hgomaa@gmu.edu

Use-case based testing methods for SPLs extend use cases, or functional models developed from these use cases, to be configurable for an application derived from a SPL [1, 2]. These methods design and customize reusable tests for a fixed set of applications derived from the SPL.

This paper describes how a feature-based testing method can be combined with a use case based testing method for SPLs, to determine what application configurations to test [2, 3] and to reduce the number of reusable test specifications needed for a set of applications selected from an SPL. Customizable Activity Diagrams, Decision Tables and Test Specifications (CADeT) is a functional testing method for a SPL to create test specifications from both the use case and feature models of a SPL. A feature-based coverage criterion is applied together with a use case - based coverage criterion to cover all features, all use case scenarios, and selected feature combinations of a SPL. CADeT reduces the number of reusable test specifications that need to be created to cover all use case scenarios of the selected applications. Further, these test specifications can be automatically selected and configured during feature-based test derivation to test a given application derived from the SPL

CADeT is based on Product Line UML-Based Software Engineering (PLUS). PLUS is a feature-based multiple-view modeling design method for software product lines. In the requirements phase, PLUS uses feature modeling to model variability, use case modeling to describe the SPL functional requirements, and a feature/use dependency table to show the relationships between features and use cases [4]

A test engineer uses CADeT to create a customizable activity diagram from each use case description in the use case model, and to associate the activities in the activity diagrams with features in the feature model. CADeT uses UML stereotypes [5] to distinguish between different granularities of functional variability in the activity diagrams of a SPL, and feature conditions to relate this variability to features in the feature model.

Next, a test engineer creates a decision table for each use case activity diagram in the SPL. The engineer maps conditions and activity nodes of the activity diagram to condition rows in the decision table. Then, the engineer traces paths in the activity diagram for each use case scenario, and then converts them into test specification columns in the table.

Next, the engineer associates features with the test specifications in the decision table, or with variation points in the test specifications. The variation point values correspond to optional or variant test steps associated with optional or alternative features.

A feature-based testing method for SPLs as in [4, 6] can be applied to determine what applications to test. Selecting a feature for an application during application derivation automatically selects the test specifications associated with that feature, and also customizes the variation points in the test specifications.

The feasibility of method was evaluated on a Banking System and Automated Toll System SPLs [6]. The results of each case study showed that the method was feasible, and could be used to substantially reduce the number of test specifications created for a SPL.

References

1. Reuys, A., et al., *Model-Based Testing of Software Product Families.* Lecture Notes in Computer Science, 2005. **3520**: p. 519-534.
2. McGregor, J.D., *Testing a Software Product Line*. 2001, SEI.
3. Scheidemann, K. *Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems*. in *10th Int'l Software Product Line Conference*. 2006. Baltimore, MD: IEEE Computer Society Press.
4. Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. The Addison-Wesley Object Technology Series. 2005: Addison-Wesley.
5. OMG, *Unified Modeling Language: Superstructure, version 2.1*. 2007, Object Management Group.
6. Olimpiew, E.M., *Model-Based Testing for Software Product Lines*, in *Department of Computer Science*. 2008, George Mason University: Fairfax, VA.

A Test-Driven Approach to Establishing & Managing Agile Product Lines

Yaser Ghanam, Shelly Park, Frank Maurer
University of Calgary
{yghanam, parksh, maurer}@cpsc.ucalgary.ca

Abstract: Test Driven Development (TDD) is an agile method that emphasizes writing tests before writing code as a means of 1) assuring the satisfaction of customer requirements, and 2) reinforcing good design habits. While the first objective is usually accomplished by acceptance tests, the second objective is achieved by unit tests. The different kinds of tests also serve as a multilevel cohesive reference of the system specifications. We propose the use of this referencing mechanism – test artifacts – to establish and manage agile product lines. Traditionally, software product lines are established through a two-phase "domain-then-application" process. Although this approach has proved successful in industry, it is usually resource intensive and risky, especially when dealing with fast changing domains. Therefore, efforts to make software product lines more agile are increasingly important as the hope is that less upfront effort will be required and risk will be reduced by utilizing agile approaches in the development of product lines. We are interested in investigating whether test artifacts can be used for this purpose within a bottom-up approach to build product lines.

In this paper, we delve into some of the issues that need to be tackled before test artifacts can be relied on as a driving force for reuse in product lines. These issues include:
1) *Establishing a framework for reuse:* Tests that have previously been produced to drive development in applications of the same domain are to be refactored and reused whenever applicable. However, determining this applicability is not straightforward since we need first to put required features in the new system within the context of previously developed ones (the family) to search for a match from existing assets.
2) *Tests comparability:* In order to crunch knowledge about whether the newly requested features correspond to the previously developed ones, there needs to be a mechanism to compare tests to realize how similar they are. This is especially tricky because we need first to clearly define what it means for two tests to be similar.
3) *Test traceability:* For an asset to be reused across different products, there has to be a mechanism to trace this reuse so that future reference is feasible. The question is how we can employ tests to trace code reuse across different products.
4) *Test refactoring:* Assuming a healthy practice of a test-driven approach, development occurs only with accordance to written tests. Therefore, when a change is to occur in the source code, all corresponding tests have to be updated. This issue becomes more incisive when considering a family of systems where change is more likely to occur and affect a wider range of source code.
5) *Test versioning:* This might not be an issue when thinking about single systems where a simple one-to-many relationship exists between the system and its acceptance tests. However, the issue becomes a quagmire in product families due to the complex many-to-many relationships that occur between applications and reusable assets.

This paper also discussed the suitability of acceptance tests and unit tests as reusable artifacts. A preliminary analysis of two systems is presented to shed light on the roles that acceptance tests and unit tests could play in the context of a test-driven product line. The initial results suggest that because developers and customers view the system differently, they give different emphasis on what is to be tested. Therefore, more is to be done to understand the multimodality of test artifacts and their appropriateness to steer reuse. Currently, we are preparing a controlled experiment to better understand the factors that contribute to how stakeholders write tests, and how developers translate these tests into design decisions. We also aim to conduct a series of experiments to investigate issues related to test-driven reusability and refactoring.

The 5th Software Product Lines Testing Workshop (SPLiT 2008)

Peter Knauber[1], Andreas Metzger[2], John D. McGregor[3]
[1]Mannheim University of Applied Sciences, p.knauber@hs-mannheim.de
[2]University of Duisburg-Essen, Andreas.Metzger@sse.uni-due.de
[3]Clemson University, johnmc@cs.clemson.edu

1. Motivation

Software product line engineering (SPLE) has shown to be a very successful paradigm for developing a diversity of similar software products at low cost, in short time, and with high quality.

Similar to the development of single software products, the key aim of testing in software product line engineering (SPLE) is to uncover the evidence of faults in the development artifacts and products.

However, significant differences between SPLE and the development of single systems exist. Those differences lead to specific challenges for product line testing:

- SPLE differentiates between domain engineering and application engineering. This raises at least two issues: Which artifacts should be tested in domain engineering and which ones in application engineering? How to facilitate the reuse of SPL test artifacts?
- The explicit definition and management of variability is key to SPLE: Yet, this raises the issue on how to perform testing activities in the presence of variability. The reusable artifacts of a SPL do not define a single software product but a set of such products. Specifically, no executable system exists in domain engineering that could be tested.
- Testing techniques from the development of single software products cannot be applied directly in the SPLE context due to the above differences.
- Efficient and effective product line testing is crucial to the success of a SPLE in an organization and it depends heavily on establishing the appropriate testing strategies, techniques, and methods.

2. Topics

The 5th SPLiT workshop, like its successful predecessors, will address open challenges of testing in an SPLE setting. Many research problems require further study, including questions such as:

1) test case design for different life-cycle stages, including module, integration, and system testing both during domain engineering and application engineering,
2) increasing test efficiency and effectiveness, e.g., by means of automated testing or reuse of test artifacts,
3) testing quality properties, like reliability or performance.

In addition to the previous workshops, SPLiT 2008 will seek contributions which discuss how we can learn from other research areas and how other areas can benefit from SPL testing results. Those areas include, but are not limited to, service-oriented architecture (SOA), model-driven development, and agile methods. As an example, due to the loose coupling of software services (e.g., Web Services) a potentially unbound number of different service compositions are possible. It should thus be interesting to see how far the solutions for handling the complexity in testing the reusable artifacts of a software product line can be applied to the complexity problem of checking the potential service compositions.

Like the previous workshops ([1], [2]) SPLiT 2008 provides an opportunity to discuss innovative ideas, setting a research agenda, and starting collaborations on diverse topics of SPL testing and related areas. To this end, SPLiT 2008 brings together both testing researchers and practitioners.

3. References

SPLiT 2006 (in conjunction with SPLC 2006) – see the workshop homepage at http://www.biglever.com/split2006/

SPLiT 2007 (in conjunction with SPLC 2007) – see the workshop homepage at http://www.biglever.com/split2007/