Embedded Systems Engineering

Ed Brinksma
Embedded Systems Institute (ESI), Eindhoven

Riding a wave of exponential growth the application of embedded technology promises to affect almost any aspect of modern life. The very fabric of society is changing as intelligence in the form of software systems is finding its way into all sorts of old and new products and services, ranging from those of existential importance - e.g. health, traffic, and energy systems - to more mundane applications for convenience or entertainment - e.g., consumer electronics and gaming. In spite of its obvious importance for the world of today, the design and engineering of high-technology embedded systems is still practiced as a craft that relies on ad-hoc methods and heuristics and the talents of (few) gifted individuals.

In the light of the above, we want to discuss how to meet one of the most important technological challenges of today, viz. how to raise embedded system design from a craft to a scientifically based engineering discipline. Among the problems to be confronted are the huge diversity of embedded systems, the heterogeneity and complexity those results from their interaction with the physical world, and the often demanding requirements regarding their reliability and performance.

In our presentation we will suggest how research on embedded systems engineering can be structured, and what interaction between academia, knowledge institutes and industry is required to advance this field, both in the Dutch and the European context.

Prof. Dr. H. Brinksma is Scientific Director and Chair of the Embedded Systems Institute (ESI) in Eindhoven, the Netherlands. He is also professor of the Formal Methods and Tool Group of the Computer Science department at University of Twente in Enschede, the Netherlands.
A Security Review of the Biometric Passport

Bart Jacobs
Radboud University Nijmegen and Technische Universiteit Eindhoven
bart@cs.kun.nl

Abstract

Many countries are currently developing a biometric passport with a chip that contains fingerprints and a facial scan of the passport holder. The regulations and technology involved will be discussed and reviewed in this talk, including the relevant protocols for authentication and secure transmission.

The speaker is member of an expert panel on biometry of the ministry of internal affairs of the Netherlands. In that context his research group at Nijmegen has received a test version of the new passport and has developed terminal-side software to communicate with the chipcard.

Bart Jacobs is Professor of Software Security and Correctness and Research Director of the Institute for Computing and Information Sciences, Security of Systems (SoS) Group at Radboud University Nijmegen, The Netherlands. He is also Professor of Design and Verification of Secure Software Systems in the Formal Methods (FM) Group of the Department of Mathematics and Computer Science at Technische Universiteit Eindhoven.
Closing the Loop

- Ensuring Testable Requirements -

drs. Aleksandar Brzic

Stakeholdership:
Communication and trust problem?

An age-old dilemma:

“Unsere Wünsche sind Vorgfühl desjenigen, was wir zu leisten imstande sind.”

‘Aus meinem Leben. Dichtung und Wahrheit.’ Teil 1-3 by Johann Wolfgang von Goethe (1811-1813)
A fundamental aspect…
Requirements and the V-Model

<table>
<thead>
<tr>
<th>User Requirements</th>
<th>Acceptance Test</th>
<th>Operational Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements</td>
<td>System Test</td>
<td>System</td>
</tr>
<tr>
<td>Architectural Design</td>
<td>Integration Test</td>
<td>Assembly</td>
</tr>
<tr>
<td>Component Development</td>
<td>Unit Test</td>
<td>Component</td>
</tr>
</tbody>
</table>

So, how do we know then?
Testable requirements

- What is a testable requirement?
- A testable requirement is a requirement that has been broken down to a level where it is precise, unambiguous, and not divisible into lower level requirements.
- But then: what is “precise”, “unambiguous” and when exactly is it “not divisible” anymore? And are they realistic and achievable?
- Therefore: standards, firm agreements, frameworks, “thought spaces” and tools for maintaining them throughout the whole project space and time are indispensable!
- Reasons: continuity, accountability, security…
Some Industry Software Quality Standards

Generic: IEEE / ISO 9000-3 SQA
Medical: FDA & GxP
Defense:

Beware: Selecting appropriate standards

- A warning first: "... standards or templates cannot in themselves provide a general structuring mechanism for requirements. Rather, ... the structure has to be developed for the particular context or problem in hand. ..." (Kovitz 1999).
- But: the common ground necessary in a project can never be reached without standards, both internal and external.
- So, what should one choose? Look at the context of the problem itself and choose related standards.
- For example, when RE-ing a SarbOx- or IFRS-compliant system, the best practices for CRM might just be irrelevant...
- Not investing time in this part of the necessary choices can be fatal!
Traceability

User Req.  System Req.  Solution Component  Unit test  Integration test  User Acceptance test

Traceability through tasks in the WBS

Plans are nothing, planning is everything (Dwight D. Eisenhower)
Falling out from the Walhalla?

“Requirements validation is difficult for two reasons. The first reason is philosophical in nature, and concerns the question of truth and what is knowable. The second reason is social, and concerns the difficulty of reaching agreement among different stakeholders with conflicting goals.” (Nuseibeh & Easterbrook 2000)

Nevertheless, it DOES pay off!

- Most CRM projects fail! (Gartner 2003)
- A “third-time-lucky” SAP implementation
- All because the requirements were incomplete, incomprehensible and probably even wrong.
- Therefore: testable requirements anyone? (We should think so…)
Better software and lower costs?  
Start improving your requirements!

Tinus Vellekoop

VVSS, Eindhoven
November 24, 2005

Effects of inappropriate req’s

• 56% defects related to requirements  
  (Source: James Martin, An Information Systems Manifesto)

• 82% effort defect repair related to requirements  
  (Source: Martin & Leffinwell)

• 44% reasons to cancel projects  
  (Source: The Standish Group, Chaos Report)

• 54% initial requirements being realised  
  (Source: The Standish Group, Chaos Report)

• 45% realised requirements being used  
  (Source: Jacobs)
Results of improvements

- 25% savings on software development
- Savings also in requirements development

- Relation between requirements and cost overrun
  - < 8% on requirements process, 70 – 200% overrun
  - 8 – 14% on requirements process, < 70% overrun

Definitions of requirements

Karl E. Wiegers:
- A statement of a customer need or objective, or of a condition or capability that a product must possess to satisfy such a need or objective. A product must have to provide a value to a stakeholder.

IEEE Std 610.12:
- 1. A condition or capability needed by a user to solve a problem or achieve an objective
- 2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document
- 3. A documented representation of a condition or capability as in (1) or (2)
Requirements structure

Business requirements (why)
User requirements (what)
System requirements (how)

Requirements processes

Change control
Version control
Requirements status tracking
Requirements tracing

Requirements development
Elicitation
Analysis
Specification
Validation

Requirements management

Requirements lifecycle
Start with improvement

- Quality of requirements (descriptions)
- Requirements administration
- Requirements Management
- Organisation of roles

1 - Quality of requirements

- Basic criteria
  (IEEE 830, IEEE 1233, ISO 9126, ...)
  - Complete - full description of functionality
  - Correct - accurate description of functionality
  - Feasible - possible to implement
  - Necessary - something users really need
  - Prioritized - indication of essentiality
  - Unambiguous - single, consistent interpretation
  - Verifiable - can be tested

- Standards for levels and types of requirements

- Requirements will be used by ...
  Requirements will be validated by ...

- The Tester’s view
  Validate requirements ⇔ Prepare logical test cases
2 - Requirements administration

Attributes of requirements
- Identification
- Version
- Description
- Priority
- Status
- Source
- Owner
- Relations
- Stability

Support changes, versions, base lining, impact, tracking and tracing

Let a requirements management tool support you

The head of the department verifies the correctness of the implementation of the change. If the check fails the procedure is started again by assigning the alteration of law. If the check passes the head of the department approves the change so the change can be used for production and calculation of assessments.
### 3 – The right time for management

Where is the dynamic?
When do requirements change?
When ... version ... status ... traceability ... impact ...

- Requirements development
- Elicitation
- Analysis
- Specification
- Validation

- Baseline
- Change control
- Version control
- Requirements status tracking
- Requirements tracing

It all starts here

### 4 - Organisation of roles

<table>
<thead>
<tr>
<th>What is there to do?</th>
<th>Who is doing what?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stakeholders analysis</td>
<td>Authorisation</td>
</tr>
<tr>
<td>Introduction</td>
<td>Requirements validation</td>
</tr>
<tr>
<td>Requirement elicitation</td>
<td>Requirements reporting</td>
</tr>
<tr>
<td>Administration</td>
<td>Base lining</td>
</tr>
<tr>
<td>Requirements analysis</td>
<td>Glossary</td>
</tr>
<tr>
<td>Teaming</td>
<td>Requirements specification</td>
</tr>
<tr>
<td>Organise co-operation</td>
<td>Requirements validation</td>
</tr>
</tbody>
</table>

How can we manage?

Requirements Analysts, Requirements Manager, Requirements Administrator
Summary

• Requirements are the key to improve quality and to decrease overrun
• Improvements pay off
• Improvements
  – Basic approach: pyramid and processes
  – Quality of requirements
  – Administration
  – Manage & Develop
  – Roles (Requirements Manager)
Motivation

- 1956 – Specifying test cases = Specify the environment and the outputs
- 1970 – Specification of test cases very important, tests should not be performed by a programmer
  - Test case documentation = Inputs, expected outcomes, execution steps
  - Documentation specifying inputs, predicted results, and a set of execution conditions for a test item

„Each test performed during this phase is documented in a set of test specifications that detail the environment used and the outputs obtained.“
Bedington 1956: „Production of Large Computer Programs“

„During the testing phase, with good documentation the manager can concentrate personnel on the mistakes in the program. Without good documentation every mistake, large or small, is analyzed by one man who probably made the mistake in the first place because he is the only man who understands the program area.“
Royce 1970: „Managing the Development of Large Software Systems“

„Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.“
IEEE Standard for Software Test Documentation

Documentation of test activities considered important from the beginning!!
What Is a "Good" Test Specification?

**Motivation**

Timea Illes
ilhes@informatik.uni-heidelberg.de

Evaluation of Present Approaches

**What is a Good Test Specification?**

- Test documentation …
  - … facilitates communication between the participants of the testing process
  - … facilitates reuse of test cases
  - … facilitates execution of test cases by unexperienced testers
  - … facilitates planning and effort estimation
  - …
  - AND: facilitates transition from requirements to test execution

**What is a Test Specification?**

… Documentation of all test cases

What is a test case?

- A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. (ISTQB)

- … a test case is a question that you ask of the program. The point of running the test is to gain information, for example whether the program will pass or fail the test. It may or may not be specified in great procedural detail, as long as it is clear what is the idea of the test and how to apply that idea to some specific aspect (feature, for example) of the product.” (Kaner, "What Is a Good Test Case?")
What Is a “Good” Test Specification?

**Motivation**

- Stakeholder & Criteria
- Evaluation of Present Approaches
- * involved in the sw development process / testing process
- * interested in (parts of) the test specification
- * e.g. test manager, tester, test automator, requirements engineer (RE)

**Criteria**

- Design & document test cases
- Testing Tasks & Activities
- Decision

- * Which logical test cases?
- * Which logical test data?
- * What automate?
- * Which parts will be affected by a change?

**Conclusion**

© 2005 Institut für Informatik Ruprecht-Karls-Universität Heidelberg

Folie 5
What Is a “Good” Test Specification?

**Motivation**

Timea Illes
illes@informatik.uni-heidelberg.de

**What is a Good Test Specification?**

**Stakeholder & Criteria**

**Conclusion**

© 2005 Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

---

**Evaluation of Present Approaches**

Tester

- Visual
- Simulatable
- Learnable
- Concise
- Semi-formal
- Repeatable
- Unambiguous
- Complete

Efficient execution

Decision

* Sequence of test cases?
* Concrete test data / test steps?
* Test passed or failed?
* Severity of defect?
* Blocking defect?

Tester performs as output

Testing Tasks & Activities

Tester requires input

Capture and compare test results

Report test results

Execute test cases

---

**Current Approaches (1)**

- **TTCN-3**
  - Tree and Tabular Combined Notation
  - Standardized test specification and test implementation language

**Key concepts**

- **Imports**
  - import of definitions in other TTCN-3 modules or other languages

- **Data types**
  - user defined data types

- **Test data**
  - templates, values transmitted

- **Test behaviour**
  - dynamic test behaviour

- **Test configuration**
  - definition of test components and ports
What Is a “Good” Test Specification?

Motivation

Stakeholder & Criteria

Conclusion

© 2005 Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

Folie 9

Current Approaches (2)

U2TP

- UML Testing Profile (OMG) based upon UML 2.0

- Key concepts
  - Test Architecture
test context, test configuration, test component, SUT, arbiter
  - Test behaviour
test case, verdict, validation action, defaults
  - Test Data
logical partitions, pattern matching

Folie 10

Current Approaches (3)

IEEE

- Standard for software test documentation
- Key concepts: Detailed documentation of the whole testing project

1. Test Plan Identifier
2. References
3. Introduction
4. Test Items
5. Software Risk Issues
6. Features to be Tested
7. Features not to be Tested
8. Approach
9. Item Pass/Fail Criteria
10. Suspension Criteria
11. Test Deliverables
12. Remaining Test Tasks
13. Environmental Needs
14. Staffing and Training Needs
15. Responsibilities
16. Schedule
17. Planning Risks and Contingencies
18. Approvals
19. Glossary

UML-based approaches

- Subsumes all approaches which derive test cases from (enriched) UML models
What is a Good Test Specification?

- **Traceable**
  - Parametrization of test case execution
  - Import of test data definitions/modules
  - No explicit traceability to requirements
  - No explicit traceability between test-cases <-> test data

- **Generalizable**
  - Test Data:
    - Wildcards, Templates

```
Template book mySpecialBook := {
  ISBN := ?,
  title := "mySpecialBookTitle",
  author := ?
  ...
}
```

```
Template bookList mySpecialBookList := {
  mySpecialBook
  mySpecialBook
}
```
What is a Good Test Specification?

**Motivation**

Timea Illes
illes@informatik.uni-heidelberg.de

**What is a Good Test Specification?**

**Stakeholder & Criteria**

**Conclusion**

© 2005 Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

---

**Evaluation of Present Approaches**

**Evaluation of TTCN-3**

**Expressive**

- Test data
  - Proprietary type system
  - Composed data types
  - No complex conditions

- Test behaviour
  - in functions or test cases
  - Alternatives
  - Conditional execution

- System states
  - Implicit (History)
  - Inquiry

- Functional test

- Testing quality requirements
  - Performance test

---

**Readable**

- Not learnable: Programming language
- Visual & simulable
- But
  - No abstraction!
  - No representation for test data
- Not concise

---

```
type record url {
  charstring protocol,
  charstring host,
  charstring file
}
```
What Is a “Good” Test Specification?

Motivation

Timea Illes
ailles@informatik.uni-heidelberg.de

What is a Good Test Specification?

Stakeholder & Criteria

Conclusion

© 2005 Institut für Informatik Ruprecht-Karls-Universität Heidelberg

Evaluation of TTCN-3

Testcase myTestCase_1 ()

myPort.send (requestURL);

localTimer.start;

alt {
  [] myPort.receive mySpecialBookList {
    localTimer.stop;
    setVerdict(pass)}
  [] myPort.receive {
    localTimer.stop;
    setVerdict(fail)}
  [] localTimer.timeout {
    setVerdict(fail)}
} /* end alt

} /* end testcase

evaluation of Present Approaches

How GOOD Are Present Approaches?

What is a Good Test Specification?

Motivation

- Stakeholder & Criteria
- Evaluation of Present Approaches
- Conclusion

UML-based approaches

IEEE

U2TP

TTCN-3

Expressive not formal not aggregatable

Notation to design the architecture of an executable test system

Describes the "maximum" information

A test focused model, allows a better derivation of test cases than the development focused model

Requirement Specification

- Use-Cases
- Model requirements from user’s the point of view
- Semi-formal
- not expressive
- incompleter

- detailed
- not formal
- incomplete

Semi-formal

formal
detailed
readable
expressive
not readable

- incomplete
- not formal
- not aggregatable

- visual
- semi-formal
- incomplete

Automatable

formal

detailed

Programm language for executable test cases

Not readable

Expressive

not formal

not aggregatable

Incomplete

not formal

not aggregatable

Incompleter

automatable

detailed

Formal
What Is a "Good" Test Specification?

Motivation

Timea Illes
alles@informatik.uni-heidelberg.de

Evaluation of Present Approaches

What is a "GOOD" test specification?

Facilitates tasks and activities of ALL stakeholders!

Conclusion

© 2005 Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

Folie 17

Good for WHOM?

What is a Good Test Specification?

Stakeholder & Criteria

Conclusion

© 2005 Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

Folie 18

What is a "GOOD" test specification?

Facilitates tasks and activities of ALL stakeholders!
What Is a “Good” Test Specification?

Conclusion

- No approach defines
  - generalization and traceability mechanism
  - prioritization mechanism for test sequences or test cases

Consequence

- No approach exists
  - for conceptual modelling test cases
  - which combines the needs of a test designer AND tester / test automator
  - which supports testing activities AND managerial activities

Evaluation of Present Approaches

- New platform for test tool evaluation
  - for test tool vendors: Evaluate own products according to defined criteria
  - for practitioners, researchers: Inquiry of test tools according to selected/entered criteria

- New platform providing current information on test related themes
  - for practitioners: Services offered by the Software Engineering Group in Heidelberg (training & coaching, conjoint research, collaboration in teaching)
  - for researchers & practitioners: Current research results (publications, talks)
testing with functions as specifications

Pieter Koopman
Radboud University Nijmegen
The Netherlands

overview

• the scene:
  automatic testing of reactive systems
    ➢ specification of properties needed
• specification of reactive systems
  ➢ transition functions
• conformance
  ➢ relation between specification and tested object
• testing conformance
  ➢ on-the-fly testing
• some examples
• other uses of the transition function
• conclusion and future work
the scene

- **testing:**
  - planned experiments with implementation to determine quality aspects
  - formal quality requirements needed
- **functional testing:**
  - focus on behaviour of (software) systems
    - relation between input and output
- **reactive systems:**
  - reaction depends on input and state
  - state determined by the history

---

specification of reactive systems

- based on FSM / statecharts
  - e.g.: machine produces Coffee or Tea after making a choice by pressing a button and inserting a coin
  - reaction on Coin depends on history

---

input: Coin
output: Coffee

initial state

state

Coin / [ Coffee ]

CoffeeBut / []

S_{coffee}

TeaBut / []

S_{tea}

partial specification: effect of Coin undefined
nondeterminism

- nondeterminism is needed if
  - system is really not deterministic
  - not the entire state is known in the specification
    - machine produces coffee if there is water and beans, but specification does not know if this condition is met

E SM: Extended State Machines

- state, input and output can be any type
  - in particular there can be arguments
specification as transition table

- pictures of statecharts are nice, but a tool requires another representation
- a table is fine for finite state machines, but not for extended state machines
  - variables causes state explosion

<table>
<thead>
<tr>
<th>state</th>
<th>input</th>
<th>output</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>OnOff</td>
<td>[]</td>
<td>Paid 0</td>
</tr>
<tr>
<td>Paid 0</td>
<td>OnOff</td>
<td>[]</td>
<td>Idle</td>
</tr>
<tr>
<td>Paid 1</td>
<td>OnOff</td>
<td>[]</td>
<td>Idle</td>
</tr>
<tr>
<td>Paid 2</td>
<td>OnOff</td>
<td>[]</td>
<td>Idle</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paid 0</td>
<td>Button</td>
<td>[Beep]</td>
<td>Paid 0</td>
</tr>
<tr>
<td>Paid 1</td>
<td>Button</td>
<td>[]</td>
<td>Paid 1</td>
</tr>
<tr>
<td>Paid 1</td>
<td>Button</td>
<td>[Coffee]</td>
<td>Paid 0</td>
</tr>
<tr>
<td>Paid 2</td>
<td>Button</td>
<td>[]</td>
<td>Paid 2</td>
</tr>
<tr>
<td>Paid 2</td>
<td>Button</td>
<td>[Coffee]</td>
<td>Paid 1</td>
</tr>
</tbody>
</table>

nondeterministic transition function

- list all possible transitions

```
spec :: State x Input → Set (State x List Output)
spec (S_tea, Coin) = { (Idle, [Tea]) }
spec (Idle, Surprise) = { (S_tea, [ ]), (S_coffee, [ ]) }
spec (S_coffee, Coin) = { (S_coffee, Coin), (Idle, [Coffee]) } 
...
```

- single function couples output and target state
advantages of functional programming languages

- needed data types can be stated clear and directly
- concise, high level functions
- generic programming yields reuse and control

\[ \begin{align*}
\text{State} & = \text{Idle} \mid \text{Paid Int} \\
\text{Input} & = \text{OnOff} \mid \text{Coin} \mid \text{Button} \\
\text{Output} & = \text{Coffee} \mid \text{Beep}
\end{align*} \]

\[ \text{spec} : \text{State Input} \rightarrow \left[ (\text{State}, [\text{Output}]) \right] \]

- \[ \text{spec Idle OnOff} = \left[ (\text{Paid 0}, []) \right] \]
- \[ \text{spec (Paid n) OnOff} = \left[ (\text{Idle}, []) \right] \]
- \[ \text{spec (Paid n) Coin} = \left[ (\text{Paid (n+1)}, []) \right] \]
- \[ \text{spec (Paid n) Button} \]
  \[ \mid n > 0 = \left[ (\text{Paid (n-1)}, \text{[Coffee]}), (\text{Paid n}, []) \right] \]
- \[ \text{spec (Paid 0) Button} = \left[ (\text{Paid 0}, \text{[Beep]}) \right] \]
- \[ \text{spec s i} = [] \]

not any statechart is a valid specification

- not every statechart can be used as specification
  - states must contain enough information
  - inputs allowed must be clear
  - transitions must be completely specified
- correct functions can always be used
testing

- SUT: System Under Test
  assumed to be black box state machine
  ➢ apply input;
  ➢ observe output
  ➢ behaves as extended state machine
  ➢ input enabled

conformance

- specification transition function
  ➢ nondeterministic extended state machine
  ➢ can be partial
    - nothing defined for some states and inputs

- implementation
  ➢ black box
  ➢ input enabled state machine

- conformance relation:
  ➢ if the specification does not cover an input for some state, anything is allowed
    - testing yields no information
  ➢ otherwise, only the specified transitions are allowed
testing conformance

- **SUT is not** conform to the specification

```
SUT

S0

S1

S2

S3

OnOff / []

OnOff / n=0; []

OnOff / n>1; []

Button / []

Button / [ Beep ]

Button / [ Coffee ]

Coin / []

Coin / [ Beep ]

Coin / [ Coffee ]

no new state: error found
```

repeat N times:

- select input allowed in current state;
- apply input to SUT and observe output;
- if output allowed by specification, compute new state;
- else report error;

**advantages**

- testing is fast and reliable
- no problems with state space explosion
- better testing by changing a parameter
- tests corresponds always to current specification

**on-the-fly testing tools**

- for these specifications: Gast
- related tools: Torx, Spec Explorer, T-Uppaal, ..
test tool Gast

- transition function
  - function in fpl Clean
  
  \[
  n \text{ Button } | n>0 / n=1; \text{ Coffee }
  \]

  spec ( n, Button ) | n > 0 = ( n-1, [ Coffee ] )

  - 1-1 mapping between statechart and function

- interface to SUT depends on situation
  - e.g.: C-API, dll, TCP/IP, ..

- test execution
  - fully automatic
  - very flexible:
    - number and length of test runs can be changed
    - input selection in a state can be controlled
    - trace information if that is desired
    - ..

some applications with Gast

- FSM in industrial context
  - over 300 states and 11,000 transitions in C++
  - error found although SUT was proven to be correct
    - the proof was correct, but incomplete (as usual)

- java-card electronic purse
  - extended state machine, much too large for a table
  - errors found in all 25 mutants (each within 1 sec)

- web-server
  - under development, first results look fine

- errors are found quickly
  - typically within seconds, but it is possible to construct errors that are very hard find
  - errors found in SUT and in specification
other uses of the transition function

- simulation / validation
- proof properties of model
  - model checking
- test properties of model
  - e.g. fairness of transitions in coffee machine
  - reachability of states
  - able to produce Coffee as output
  - ..

---

testing fairness in coffee machine

\[
\text{fair } s \ i \ (o,t) = \text{value } s + \text{value } i \ =\ = \text{value } o + \text{value } t \\
\text{fair spec } s \ i = \text{fair } s \ i \ \text{For spec } s \ i \\
\text{Start} = \text{test (fair spec1)}
\]
conclusion

• manual testing is good
  ➢ very flexible
• script based testing is better
  ➢ executing tests is fast and accurate
  ➢ easy to repeat tests
• model based on-the-fly testing is best
  ➢ tests always conform the current specification
  ➢ use a small specification instead of large test suites
  ➢ number of tests controlled by parameter
• simple formal specification needed
  ➢ testing helps to improve SUT and specification
  ➢ specification also used for validation and verification

future work

• applications
  ➢ web-servers
  ➢ large embedded systems
  ➢ Dutch biometric passport
    - contains smart card with biometric data
  ➢ ...
• theory/tool development
  ➢ using UML specifications
  ➢ determine quality of tests
  ➢ determine quality of system after tests
  ➢ time in specifications and tests
  ➢ asynchronous communication in model
  ➢ ..
Maintainable Tests in Rational Functional Tester

LaQuSo – 24 November 2005

Wie is Marc van Lint?

Functie: Rational Technical Specialist
Focus: Kwaliteit in Software Ontwikkeling

Marc van Lint
IBM Nederland N.V.
Johan Huizingalaan 765
1066 HV Amsterdam
Telefoon: +31-20-513-3065
Mobiel: +31-6-53244587
E-mail: marc_van_lint@nl.ibm.com
Discover, develop, and deploy assets

- **Discover business & technology assets**
  - Business priorities
  - Requirements
  - Middleware and software assets

- **Develop at the speed of business**
  - Rapid application development
  - Model-driven architecture
  - Asset-based development
  - Direct-to-middleware productivity

- **Deploy to closed-loop environments**
  - Automated applications deployment
  - Streamlined composite application management
  - Direct-to-operations productivity

---

Key Principles for Business-Driven Development

- Adapt the process
- Balance stakeholder priorities
- Collaborate across teams
- Demonstrate value iteratively
- Elevate the level of abstraction
- Focus continuously on quality
The IBM Software Development Platform
A complete, open, modular, and proven solution

IBM Products

IBM Software Group | Rational software

Windows
UNIX / Linux

ECLIPSE

Customer Extensions

3rd Party ISV Tools

IBM Products

IBM Software Group | Rational software

Page 37
Rational TestManager: The control center

- **TEST INPUTS**
  - Requirements
  - Spreadsheets
  - Models

- **PLAN & DESIGN**
  - Test Configurations
  - Test Cases
  - Test Iterations

- **EXECUTION**
  - Cross-platform
  - Any kind of test: unit, functional, load, manual

- **INPUT**

- **VIEW RESULTS**
  - Pass
  - Fail
  - Generate change requests
  - Coverage reports

- **Rational TestManager**

- **Many tests at same time**

- **Requirements**

- **Spreadsheets**

- **Models**

- **Test Configurations**

- **Test Cases**

- **Test Iterations**
Use Case to Test Suite

Use Case  |  Test Plan  |  Test Suite
--|---|---
- Test Case A - Test Case B - Test Case C - Test Case D

Rational Manual Test (RMT)

- Rich text editing including images, attachments, and granular and high level verifications
- Import content from Word or Excel
- Automated data entry and verification capabilities
- Custom properties to capture data and metrics to fit any process
- Drag and drop reusable test steps onto Reuse palette
- Single point updating of reused content
IBM Rational Functional Tester (RFT)

- Eclipse based
- Environments:
  - Web/Java
  - MS VS.NET
- Code:
  - Java
  - VB.NET
- Full IDE Integration
- Data driven

RFT: Data driven testing

- Separation of program and data
- Easy testing on exceptions

(TestManager) Data Stores -> Test Script -> Program
RFT: Shared Object Library

- Separation of program and objects
- Change object updates all programs
- Object has various properties and weight for recognition
RFT: ScriptAssure™

Version 1.0

Tester Sees

Version 2.0

Customer Log On

Property | Value
---|---
class | Html:INPUT
id | button1
name | userLogin
type | submit
value | Log On

Property | Value
---|---
class | Html:INPUT
id | button1
name | userLogin
type | submit
value | Log On

Granularity

Recognized = 0
Not found = 100

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Recognized</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>.id</td>
<td>90</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>.type</td>
<td>95</td>
<td>x</td>
<td>100</td>
</tr>
<tr>
<td>.value</td>
<td>100</td>
<td>x</td>
<td>100</td>
</tr>
</tbody>
</table>

Maximum acceptable recognition score: 10000  
Last chance recognition score: 20000  
Ambiguous recognition scores difference threshold: 1000  
Warn if accepted score is greater than: 5000

0 1K 5K 20K STOP
Conclusion

- All ‘traditional’ techniques can be applied
  - Data driven testing
- Rational Functional Tester can utilize a shared object library
  - Centralized object maintenance
  - Object attribute includes recognition weight
- ScriptAssure™ is integrated recognition intelligence
  - No additional programming
- Better control on script maintenance
  - Amount of maintenance
  - Corrections according to planning

www.ibm.com/software/rational
Pathcrawler - tool for automatic generation of path tests, combining static and dynamic analysis

Nicky Williams, Bruno Marre Patricia Mouy and Muriel Roger
CEA/Saclay, DRT/LIST/SOL/LSL, 91191 Gif sur Yvette, France

Abstract

PathCrawler is a prototype tool for the automatic generation of test-cases which are guaranteed to exhibit all possible behaviours, i.e. feasible execution paths, of the program under test. This program must be a sequential program written in an imperative programming language such as C and its source code must be available. PathCrawler is based on a novel combination of code instrumentation and constraint solving which makes it both efficient and open to extension. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions demanded by the use of heuristic algorithms in function minimisation and the possibility that they fail to find a solution.

1. Introduction

Rigorous testing of delivered software, by its implementers or by external certifiers, is increasingly demanded, along with some quantification of the degree of confidence in the software implied by the test results. The reasons for this include the increase in the deployment of embedded software systems and in the re-use of software components. This sort of testing cannot be based on a restricted set of hand-crafted test objectives or use-cases, which may have to be manually updated if the software requirements change. Testing must be made as automatic as possible, with automatic generation of a large number of test-cases according to a well-justified selection criterion.

We present the PathCrawler tool for the automatic generation of test-cases satisfying the rigorous 100% feasible execution paths criterion. In the following section we compare Pathcrawler to other work on automatic test-case generation. We then give an overview of our approach and describe its principal stages: Instrumentation, Substitution and Constraint Solving. We describe the current status of the implementation and present some performance results. In conclusion, we discuss the application of PathCrawler to different types of program and describe work in progress.

2. Related Work

There has been much research on the automatic generation of structural test-cases but many techniques do not scale up to full coverage of realistic-sized programs, mainly because they were not actually designed to generate complete test sets guaranteeing full coverage. Instead, most previous work addresses the problem (called the Test Data Generation Problem (TDGP) in [4]) of finding data to cover a "test purpose" in the form of a particular node, branch or path of the control flow graph.

Static approaches to test-case generation [2][3][11] typically select a path from the control flow graph covering the test objective, derive the path predicate as a set of constraints on the input values and then solve these constraints to find a test-case which activates the path. In theory, symbolic execution can be used to construct the path predicate. However, in practice symbolic execution encounters problems in the detection of infeasible paths (notably in the case of loops with a variable number of iterations), the treatment of aliases and the complexity of the formulae which are gradually built up.

Dynamic approaches [1][4][6] avoid the problems of symbolic execution by dispensing with the path predicate and using general heuristic function minimisation techniques to modify the input data so that the test objective is covered. The first set of input data is arbitrarily selected and the program is instrumented so as to indicate the branches taken and evaluate their "distance" from the test objective. Function minimisation must reduce this distance to zero. The disadvantages of these techniques are that they may need a great many executions before a test-case is found, they may fail to find a test-case even when one exists and they do not terminate if the desired path is actually infeasible.

As we address a different problem to that of most previous work, we adopt a different solution. Our
objective is the automation of testing with full structural coverage. PathCrawler is based on the most rigorous structural coverage criterion: 100% coverage of feasible execution paths. However, its test generation strategy can be modified to relax this criterion in a disciplined way if there are too many feasible execution paths in the program to be tested. The TDGP is not the best formulation of the problem of test-case generation for full structural coverage. We do not need to construct the control flow graph, enumerate all the paths in the graph, many of which will be infeasible, and search for a test for each. Instead, we iteratively cover "on the fly" the whole input space of the program under test. This is an extension of the idea sketched out in [10] but we apply it to path coverage instead of branch coverage and we do not risk leaving feasible paths uncovered by limiting exploration of each previous path predicate to only one prefix.

Like the dynamic approaches to test data generation, PathCrawler is based on dynamic analysis, but instead of heuristic function minimisation, it uses constraint logic programming to solve a (partial) path predicate and find the next test-case, as in the approaches based on static analysis. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions demanded by heuristic algorithms used in function minimisation and the possibility that they fail to find a solution.

3. Our approach

Our approach (see Figure 1) starts with the instrumentation of the source code so as to recover the symbolic execution path each time that the program under test is executed. The instrumented code is executed for the first time using a "test-case" which can be any set of inputs from the domain of legitimate values. The symbolic path which we recover is transformed into a path predicate which defines the "domain" of the path covered by the first test-case, i.e. the set of input values which cause the same path to be followed. The next test-case is found by solving the constraints defining the legitimate input values outside the domain of the path which is already covered. The instrumented code is then executed on this test-case and so on, until all the feasible paths have been covered.

4. Instrumentation

The instrumentation stage is an automatic transformation of the source code so as to print out the symbolic execution path, i.e. a sequence of assignments and satisfied conditions on C variables referenced by the program. These include scalar variables and access paths (containing e.g. element indices, pointer de-referencing, pointer arithmetic,…) for elements of structured data. In the rest of the paper we will use the term "variable" to refer to both scalar variables and elements of structured data.

A trace instruction is automatically inserted after each control point, i.e. sequential block of instructions or branch of the source code. A table is automatically generated to give the sequence of assignments or branch condition corresponding to each control point. This table is used generate the symbolic execution path corresponding to the recorded trace.

The instrumentation is implemented using the CIL library [9]. Certain source-code statements are decomposed, notably multiple conditions (which reinforces our test criterion, bringing it close to all-paths combined with MC/DC). Along with pointers, the C language offers alternative notations to access elements of structured data but in our trace instructions, all data access paths are represented in a canonical form. This rewriting of access paths,
which is purely syntactic, simplifies the substitution stage.

5. Substitution

A path predicate is a conjunction of constraints expressed in terms of the values (at input) of the input variables. However, the symbolic conditions output by the instrumentation of the conditional statements in the source code may be expressed in terms of local variables (or intermediate values of input variables, which we will also refer to as local variables). The substitution stage of our approach carries out the projection of these conditions onto the values of the inputs. The sequence of statements output by the execution of the instrumented program is traversed and each assignment is used to update a "memory map" which stores the current symbolic value of each local variable in terms of the input values. When a condition is encountered, all occurrences of local variables are replaced by their current symbolic values. The resulting list of conditions is the path predicate.

Because we analyse a single, unrolled, path, we do not need to use the SSA form used in [2] and can treat aliases (two or more ways of denoting the same memory location) with relative ease, as we now explain.

When the same memory location is denoted in different ways in a program then these different names for the same memory location are called "aliases". In the example code fragment of Figure 2, for execution paths for which the condition in line 3 is true, *pt is an alias for tab[x] after execution of the assignment in line 4. Unlike classical static analysis approaches, we do not have to represent more than one possible value for pt in line 7 because we treat one execution path at a time. We simply look up the current symbolic value of pt in the memory map.

However, aliases do pose a problem for us when a variable access path contain the names or access paths of other variables. In the code fragment in Figure 3, the array element whose value is updated in line 5 depends on the value of x and consequently the value of tab[y] in line 6 could be 8, 4 or 10, depending on the values of x and y. If x = y then the branch condition for line 5 is 8 < z. If x ≠ y and y = 0 then the condition is 4 < z. If x ≠ y and y = 1 then the condition is 10 < z. The path predicate for any path in which this condition is satisfied should therefore contain the following disjunction to represent this condition: (x = y ∧ 8 < z) ∨ (x ≠ y ∧ y = 0 ∧ 4 < z) ∨ (x ≠ y ∧ y = 1 ∧ 10 < z). Note that each disjunct is made up of one condition which is the interpretation of the branch condition in the source code and one or more other conditions on input values. Let us call these other relations on input variables which lead to different symbolic values for the variables in a branch condition "alias relations". Note that among the theoretically possible alias relations in such a disjunction, some may not be consistent with the legitimate input values or the rest of the path predicate.

Instead of treating path predicates containing such disjunctions, we choose to treat separately the paths arising from each disjunct. In our example, we therefore consider that the execution path in which the condition in line 6 is satisfied has up to 3 different predicates. We insert into the predicate just the alias relation effectively satisfied by the inputs in the test case whose execution gave rise to this path, along with the corresponding interpretation of the path condition. The test case in which x = 1 and y = 1 would therefore result in the predicate x = y ∧ 8 < z. Our test-case generation process naturally leads to the exploration of the other possible alias relations and corresponding path conditions.

Assignments, such as that in line 5 in our example, in which the variable name is indexed by another variable name (which does not have a constant value) pose the problem of how to update the memory map. The memory map must be enriched in order to treat such assignments. The first extension is to number all assignments in the execution path so as to determine their order. On each update of the memory map, the number of the assignment is stored along with the symbolic value. Moreover, in the case of an assignment to a data structure element whose access path is indexed by another variable, we determine the value of the index for the test set which gave rise to the execution path. It is this element which is updated in the memory map but its value now stands for the value of all other elements which satisfy the same alias relation. To ensure that this is the case, we store in the memory map, along with the

<table>
<thead>
<tr>
<th>void f (int x, int y, int tab[])</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *pt</td>
<td>(2)</td>
</tr>
<tr>
<td>if (x &lt; 2)</td>
<td>(3)</td>
</tr>
<tr>
<td>pt = &amp;tab[x]</td>
<td>(4)</td>
</tr>
<tr>
<td>else</td>
<td>(5)</td>
</tr>
<tr>
<td>pt = &amp;tab[y]</td>
<td>(6)</td>
</tr>
</tbody>
</table>

**Figure 2 : first alias example**
we therefore store for the element the path predicate. By carefully taking account of this information when evaluating the current assignment must be stored resulting from the test case in which $x = 1$ and $y = 1$, we therefore store for the element $\text{tab}[1]$ the information that the symbolic value of the index is $x$. In any future updates of the memory map entry for this element, the different symbolic values used for the indices in past assignments, and the number of the most recent assignment employing each such symbolic index value, must also be memorized. Furthermore, any alias relations which condition the evaluation of the current assignment must be stored (imagine the assignment $\text{tab}[\text{tab}[x]] = 8$!)

By carefully taking account of this information when looking up values in the memory map, we can correctly establish the alias relations and add them to the path predicate.

### 6. Test Selection and Constraint Solving

The starting point of the test generation process is the input domain of the program under test. This is the set of all legitimate input vectors, i.e. combinations of values of the different input variables. By input variables, we mean all scalar variables or elements of structured data whose value may be read during execution of the program under test without having previously been assigned by the program. This may include global variables and those referred to using pointers. The set of input variables may vary with the execution path and is difficult to determine precisely using static analysis. This is why PathCrawler currently generates a set of possible input variables which may include some which are not, in fact, input variables for any feasible execution path. The user can eliminate such variables from the set. This may include providing an upper limit on the possible size of certain arrays. PathCrawler also asks the user to define the precise set of legitimate input vectors. By default, this is the cartesian product of all values within the C type of each input variable. However, the set of input vectors to be used during testing may be much smaller than this. The possible values of a particular input variable may in fact be far fewer than those allowed by the C type. Moreover, there may be preconditions on combinations of input values which must be respected, either to avoid errors at execution due to e.g. division by zero, or just for the algorithm implemented by the program under test to be correct.

The user can define the legitimate range of each input variable and any preconditions on values of sets of variables, using a limited form of universal quantification if necessary.

The first test-case $t_1$ is chosen within a selection domain $SD_0$ which is just this input domain of the program under test (see Figure 4). From the execution of $t_1$, we derive the corresponding path predicate $PP_1$. In order to cover a new path, we have to generate test inputs from the difference, $SD_1$, of $SD_0$ and the domain of $PP_1$. If $SD_1$ is empty, this means that there are no more paths to cover. Otherwise, we can generate a new test-case $t_2$ from $SD_1$, which exercises a new path whose predicate is $PP_2$. This process is repeated until an empty selection domain $SD_n$ is reached, in which case we have covered every feasible path of the program under test.

Each path predicate $PP_i$ is the ordered conjunction of the number $p_i$ of successive conditions $C_{ij}$ encountered along the corresponding path:

$$PP_i = C_{i,1} \land \ldots \land C_{i,p_i}$$

The negation of each path predicate $PP_i$ is just the disjunction of all the prefixes of $PP_i$ with the last condition negated:

$$\neg PP_i = \neg C_{i,1} \lor \bigvee_{m=2..p_i} (C_{i,1} \land \ldots \land C_{i,m-1} \land \neg C_{i,m})$$

Note that each term of such a disjunction is a conjunction of conditions corresponding to a (possibly infeasible) path prefix which is unexplored at the $i$th step of our selection strategy.

To find a solution in each selection domain $SD_i$, we choose to solve the longest feasible conjunction in $\neg PP_i$ which we call $\text{Max}C_i$. If all the conjunctions in $\neg PP_i$ are infeasible, the longest unsolved feasible conjunction, $\text{Max}C_{i-1}$, in $\neg PP_{i-1}$ is tried, and so on. Our strategy corresponds in this sense to a depth-first construction of the tree of feasible execution paths.

Test selection and constraint solving are implemented in the Eclipse constraint logic programming environment [12]. Note that solving non-linear constraints is decidable only for data
types with finite domains, such as integers. However, current research [7][11] holds the promise of decidable and precise constraint solving for floating-point numbers too. Solving constraints over finite domains is NP-complete in the worst case but we base our work on heuristics developed for test-case generation problems [3][5] which display low complexity in practice. In the case of data-structures whose size may not be the same in all the test cases, constrained variables representing the elements of the data-structure are defined only as needed. Our "labelling" heuristic (used to generate and test values after constraint propagation) is to choose dimension values as low as possible. This has the advantage that we are sure to generate tests for empty data-structures (where they are allowed), whose treatment is often a source of bugs. Moreover, as there is often a link between data-structure dimensions and the number of loop iterations, smaller data-structures can result in fewer superfluous test cases for the k-path criterion. For variables other than dimensions, labelling uses a random generator, biased towards the middle of the variable's domain after constraint propagation.

An advantage of our test generation strategy is that we only analyse feasible path predicates. Of course during the search for MaxCm, we may construct other path predicate prefixes which turn out to be unsatisfiable, but this is always due to the negation of the last condition. We make use of this property when selecting the next variable for labelling. Moreover, when a path predicate prefix has no solution, the strategy does not construct or explore any path predicates starting with this prefix.

7. Status

Our approach is applicable to all sequential programs coded in an imperative language and the prototype has been implemented for C. The only parts of ANSI C for which we have not yet had time to implement the treatment are function pointers and recursive functions.

Our test generation strategy has an extremely efficient implementation. This is because we can use the backtrack mechanism and stack in Eclipse to effectively store the symbolic variable values and constraint store resulting from the partial path predicate for each prefix of each treated path. This avoids recalculating them when treating another path which has the same prefix.

We tried PathCrawler on three well-known examples from the testing literature: TriType, Bsort and Sample. Given the sides of a triangle, TriType carries out a series of tests on them to classify the triangle. It has no loops and only 14 feasible execution paths but is interesting because the path predicates include simple arithmetic expressions and not just inequalities as in the other examples. Bsort is a bubble sort containing two nested loops, one iterating over all the elements of the array to be sorted and the other over the elements after the current one. Sample compares the content of two arrays to a reference value in two successive loops, each with a fixed number of iterations of the length of the array. We also describe in [13] our experiments with the Merge program which fuses two ordered arrays to produce another sorted array and contains many infeasible paths. For this program, we defined a maximum number, k, of loop iterations (see Section 8). We generated the tests 10 times for each program, in order to evaluate the variation caused by our random labelling heuristic. Table 1 shows the number of tests, number of infeasible prefixes, mean execution time in seconds and variation in the execution times over 10 runs for these programs.

<table>
<thead>
<tr>
<th>program</th>
<th>array dimn.</th>
<th>tests</th>
<th>Infeasible prefixes</th>
<th>mean exec. time</th>
<th>min. exec. time</th>
<th>max. exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriType</td>
<td>-</td>
<td>14</td>
<td>3</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>Bsort</td>
<td>0 - 5</td>
<td>153</td>
<td>349</td>
<td>1.16</td>
<td>1.14</td>
<td>1.17</td>
</tr>
<tr>
<td>Sample</td>
<td>4</td>
<td>241</td>
<td>0</td>
<td>0.27</td>
<td>0.22</td>
<td>0.29</td>
</tr>
<tr>
<td>Merge</td>
<td>k = 5</td>
<td>-</td>
<td>337</td>
<td>0.78</td>
<td>0.75</td>
<td>0.81</td>
</tr>
<tr>
<td>Merge</td>
<td>k = 10</td>
<td>-</td>
<td>20993</td>
<td>15357</td>
<td>116</td>
<td>-</td>
</tr>
</tbody>
</table>

8. Further work

Some programs have so many execution paths that path testing is infeasible, even when test inputs are automatically generated and an oracle program is available. A combinatorial explosion in the number of execution paths of a program can have several causes. We are studying these causes in order to design and implement new test strategies which keep the number of tests reasonable. Fortunately, PathCrawler's test generation strategy can easily be modified to take into account information obtained either statically (e.g. from specifications or static analysis of the source code) or dynamically (e.g. from further instrumentation).

One cause of a combinatorial explosion in the number of execution paths is the presence of loops with a variable number of iterations. Strict path testing demands an individual test for all paths which differ only in the number of iterations of a certain loop. This is why the k-path criterion is often used in practice. This allows the user to define a limit, k, to the maximum number of iterations of this sort of
loop in tested paths. In [13], we show how PathCrawler was easily modified to implement this strategy and generate a reduced number of tests. The instrumentation was modified to indicate which conditions were loop heads and constraint solving was modified to take these annotations into account.

Function calls also cause a large number of execution paths, some of which may only vary in the path taken within a called function. We currently treat function calls by classic in-lining techniques. By annotating the conditions in called functions, the exploration of different paths in these functions could be restricted. Another solution which we are currently investigating is the use of specifications of called functions as “stubs” in integration testing [8].

Finally, reactive software, in which the system is first initialized and then the same program is called repeatedly to process the new input data arriving in each cycle, poses another problem for path testing. Should path testing be limited to one cycle, or should a “path” be interpreted as the sequence of paths taken in several successive cycles? In such programs, the current state of the machine is usually updated in each cycle and stored in static or global variables for use in the next cycle. We can only limit path testing to one cycle if the user can characterize (in the definition of the input domain) all the possible states at the beginning of a cycle. This is not usually the case. We are currently studying how best to modify PathCrawler’s strategy in order to adapt it to this type of software. This is particularly important in the case of one potential application of PathCrawler: the automatic generation of test-cases for the measurement of worst-case execution time, which is another subject of our current investigations [14].

10. References

[8] P. Mouy, Vers une méthode de génération de tests boîte grise “à la volée”, In Approches Formelles dans l’Assistance au Développement de Logiciels (AFADL’04), June 2004, Besançon, France
[14] N. Williams, WCET measurement using modified path testing, In Proc. 5th International Workshop on Worst Case Execution Time (WCET) Analysis, Palma de Mallorca, Spain, July 2005
An Architecture Process for Repeatable Design
Stef Joosten\textsuperscript{1,2}, Rieks Joosten\textsuperscript{3}

\textsuperscript{1}Ordina \hfill \textsuperscript{2}Open University of the Netherlands \hfill \textsuperscript{3}TNO Informatie- en Communicatie Technologie

stef.joosten@ordina.nl \hfill stef.joosten@ou.nl \hfill H.J.M.Joosten@telecom.tno.nl

Abstract
Architects face the challenge to make their work more concrete in the eyes of their clients, their users and other stakeholders. Their mission is to create a coherent and consistent structure of applications, systems, and business processes that satisfies the rules and requirements of the business. However, members of an architecture team sometimes get caught in the middle of complex terminology discussions, deadlines and extremely large amounts of design artifacts. So how are they going to deliver useful results for the business? This paper presents a software tool that signals inconsistencies and incompleteness in business, application and infrastructure architecture. An architecture team can monitor its collective work in real time, allowing architects to remove the last inconsistency. Besides, the software provides project managers with an objective instrument to monitor architecture projects.

1. Launching Business Initiatives

After putting men on the moon, NASA wanted to intensify space travel. She was in dire need for more repeatable, safer, more reliable and cost efficient means to make frequent trips to space. That is why the space shuttle program was developed. Information Technology is in a similar situation. In order to make further progress, IT must become more repeatable, safer, more reliable and cost efficient. Rather than managing every IT project individually (comparable to Saturn missions), organizations conduct IT programs (comparable to the space shuttle program) to make IT more manageable, less costly, and more predictable. Organizations that depend on continuous innovation must launch business initiatives at ever shorter time intervals. This justifies a substantial effort to turn innovation a repeatable process. That is why organizations are willing to invest in architecture. Scientific developments in this area focus on patterns [e.g. 1, 2], enterprise frameworks [e.g. 3], or methodology [e.g. 4]. Our approach is to study the architecture process, referring to the work of architects that is concerned with developing satisfactory and feasible system concepts, maintaining the integrity of those system concepts through development, certifying built systems for use, and assuring those system concepts through operational and evolutionary phases [5]. An architecture process serves as a booster rocket, fuelling the innovation process to perform reliably when launching new business initiatives.

This article focuses on the architecture process, which we view as a process of rule making and monitoring. We will show how these rules can be used to monitor the architecture process in real time. This brings the idea of IT architecture from a description mechanism towards a control mechanism. Only then architecture might do for IT what the electricity plug has done for home appliances: more freedom to bring new ideas to larger markets with a better chance of success.

Essential ingredients of architecture are business rules, i.e. rules that are verifiably true or false, universally valid in a particular context, and provide relevant information to the business when violated. Throughout this paper, we use the word rule or business rule in this particular meaning. Otherwise we use the word principle or guideline.

This paper starts with a discussion on the architecture process. We then discuss the role of an architecture monitoring tool. The tool we have built uses the ArchiMate\textsuperscript{1} language in the role of “standardized electricity plug”. Then we discuss an experiment conducted with that tool. The results show that the rules, which govern architecture, can be used to build architecture checkers in a generative way. That provides IT-governance with a concrete instrument for checking architecture compliance.

\textsuperscript{1}The ArchiMate project (http://ArchiMate.telin.nl) was partially funded by the Department of Economic Affairs and delivered on December 31, 2004.
This paper adheres to the architecture definitions from the IEEE Recommended practice 1471-2000 [5].

2. Designing Repeatably

The quality of a large design depends largely on the level of coordination an architecture team can achieve. Members of an architecture team spend much of their time trying to match design decision with business requirements, trying to fix solutions in the infrastructure, trying to solve difficulties with legacy applications, trying to avoid inconsistencies, communicating with stakeholders, trying to keep users involved, and so on. Making a large design consistent and complete often requires many meetings, peer reviews, and lots of interviews and workshops. Coordination is the name of that game.

In our analysis we have identified the following (groups of) stakeholders:

- architect: wants clarity, less discussion and more results;
- Architecture team: wants a concrete result in a consistent, buildable way, with support of all stakeholders;
- Project leader: wants to manage a team of architects;
- Acquirer (e.g. an executive who has assigned an architect to a project): wants assurance, low cost, control;
- Customer: wants fast, flexible and fine services.

The purpose of architecture is to accelerate and improve the innovation process such that new business initiatives can be launched routinely and reliably. The research focuses on the question how an architecture process fosters repeatability.

3. Managing Architecture

Architects face the challenge of structuring complex situations. They must bring clarity and reduce perceived chaos by providing simple icons and metaphors that inspire stakeholders. They are put to the challenge of curbing that complexity. This broader challenge must be understood before turning to solutions. To that end, we have studied the architecture process.

If architecture makes innovation into a repeatable process, and a repeatable innovation process is required to launch business initiatives, a strong resemblance with the space shuttle program emerges. The large fuel tank corresponds to the innovation process, where architecture and management serve as booster rockets (figure 1). The launching of the shuttle itself represents the launching of business initiatives, which is done repeatedly, reliably, and relatively cost efficient. The entire system is designed to bring large numbers of business initiatives into orbit.

In our analysis we have identified three levels of architecture: the project, the program and the corporate level. Architects provide concrete form and meaning in all three levels. In IT projects they create innovations that affect both the organization and information technology. Depending on the particulars of each project, various kinds of designers are involved, such as business designers, process designers, application designers, infrastructure designers, etcetera. One level up, at the program level, architects make rules and principles for the purpose of coordinating individual project efforts. Here, architects study commonalities of large numbers of projects, enforce standards, create reference models, collect best practices in the domain, and disclose their work to all stakeholders involved in projects within that program. On the next level, corporate architects set standards, devise rules that implement governance principles (such as IFRS, the Sarbanes-Oxley act of 2002, safety regulations, etc.), implement corporate policies, etcetera. The project, the program and the corporate levels correspond to the operational, tactical and strategic management levels of the innovation process.

Architecture can be understood as a process of rule making and rule monitoring on all three levels.

---

2 The idea of an architecture process and a management process that support the innovation process from two sides is due to Tinus de Gouw, who currently works with Rabobank.
On the corporate level, architecture provides the rules and principles that are valid throughout the organization. Within each program, rules are defined that are valid throughout the program but not beyond. Each project must abide by the rules of the program and the corporate rules. Besides, every project may have its own architecture, setting particular rules within the project. In this analysis, architects require a rule base in which a rule is valid within its particular context.

Any omission and any violation of a rule made by an architect may yield problems when the design is realized. It always takes extra time, but may also cause rework or even redesign, leading to possible setbacks in the innovation process. Thus, violations of rules pose a direct threat to the repeatability and reliability of the innovation process. If designs are guaranteed to be free of architectural violations, this increases repeatability of innovation, and decreases the risk of launching new business initiatives.

In order to obtain flawless designs, we need a mechanism to signal violations. This requires to know which rules apply to a design, a mechanism to compute signals on the basis of violations, and a way to communicate those signals to a stakeholder with the authority to act upon each signal. Computer support is needed here. There are many different rules that are valid within many different contexts in an organization, so it is not reasonable to manage those rules ‘by hand’ and expect no mistakes. There are many different projects and a vast amount of design artifacts, so it is far too much work to take out all rule violations without the help of computers. These requirements inspired us to build an architecture checker.

As a result, an architecture process can be implemented as depicted in figure 2. If all design changes are fed into a repository, a checker can produce signals and feed them back into the architecture process. A signal confronts an architect instantaneously with design decisions of his or her peers. The mechanism is limited to a signaling function only. Enforcement is left to the individual style of each project.

Our analysis shows that designers need more than tools for drawing and software generation. Besides the available tools, a checker to monitor architecture is useful to keep team members aligned with the rules of the business.

4. Checking the Rules

The architecture checker that was built has a simple structure (figure 3). A repository is the foundation. It contains information about business processes, roles, applications, services, nodes, communication paths, etcetera, according to a structure described in the ArchiMate project. This choice was made because the ArchiMate architecture language has a reknown status in the Netherlands and is acknowledged by Dutch professionals throughout science and industry. The ArchiMate reference manual [7] provides an accurate description of the language structure in terms of a metamodel. Semantic rules however, are described in natural language. Most of these rules describe multiplicity restrictions, i.e. omissions and ambiguities that might arise from design errors. The repository (written in MySQL) satisfies the ArchiMate structure (the metamodel) and the rules of ArchiMate have been translated into a software component (written in PHP) that checks for violations (the service layer) and presents them as signals in a browser (the presentation layer).

Architects gain access to the repository and checker by means of a browser. The repository allows multiple users, so any changes made by one architect are visible for the team members. The visualization component is currently (at the time of writing) being installed at the Telematica Institute in Enschede. The repository and checker have been built at Ordina. The design is such that later extensions can be made without excessive effort.

Designers can use the checker by inspecting and analyzing signals from the checker and changing the design (as represented in the repository) accordingly. In doing so, new signals may arise from the checker.
By dividing the total design space among themselves, architects can distribute the work. For instance, one might concentrate on the business architecture, another on the application architecture and a third on the infrastructure. If for example, a team member defines a new service, an omission arises in one of the tables in the repository, saying that a node is required on which to run that service. When an application is defined to use that service, a signal is risen when there is no interface to make that service available. These examples (and all others) show how an architecture checker provides architects with useful information to complete or correct their work. The repository stores concrete design choices in tables, such as the assignment of application components to processes, business roles to business interfaces, network components to software components, etcetera. Whenever a signal occurs, it is up to the designer to determine the meaning of that signal (diagnosis). The checker provides the signals only, relating them to the particular rule being violated. When a team is done and all signals are resolved, the checker guarantees that the design satisfies all of ArchiMate’s rules.

5. Experimenting with the checker

The first experiment was carried out on September 12th 2005. The purpose of this experiment was to gain insight in practical questions: Can architects grasp the idea quickly enough? Does the tool impose unreasonable restrictions? What can an architecture team achieve in a limited amount of time? Is the software sufficiently robust? And most importantly: do architects feel that this type of tool is useful?

We picked three experienced architects, one of which was knowledgeable with Archimate. We confronted the subjects with the design of a (fictitious) insurance company, ArchiSurance [6]. Before the experiment, the ArchiSurance design was translated literally from Archimate documentation into the repository. The team was asked to prepare by studying an Archimate primer [6] and the ArchiSurance case contained in that primer. The experiment consisted of resolving all signals detected by the checker in one hour. Since the checker was new to all team members, a short oral instruction was provided just before the experiment. Each team member was given one part of the design space as his own responsibility. By keeping the preparation down to an absolute minimum, the experiment provided a good indication about the threshold of use.

During the experiment, it took the team about 15 minutes to get used to what the tool showed them and to get going. After an hour, the team had investigated twenty signals and resolved thirteen. Team members would typically trace a signal straight back to the original design, and negotiate who would make the necessary adjustments.

In a retrospection session, both the architecture process and the tool were experienced positively. Team members focused their attention especially to the rules, questioning whether the right rules were being checked. They experienced the nature of ArchiMate’s rules to be too general. Control questions showed that the subjects were very much aware of what they were doing. For example, they were able to place the checker flawlessly in the upper left area of figure 1 (without having read this paper…) The fact that the entire design was represented in a repository allowed them to get down to work straight away. None of the team members had felt the urge to address terminology of definitions underlying the architecture. The primary contribution was seen in the mutual coordination among architects in a team.

6. Results

The results of the experiment show that the checker has supported the team as intended. On the basis of these results, more experiments and more specific experiments will be conducted in the near future.

The architecture checker means different things to different stakeholders. Designers have an instrument to coordinate their work. They can freely invent their designs, but their work may yield signals elsewhere. The discussions that arise are concrete, since they are based on concrete signals. Also, these discussions are necessary in order to resolve signals. The experiment showed that these discussions are necessary, relevant and to the point, indicating that the checker indeed helps to avoid abstract, pointless discussions.

For the team as a whole, the checker results in a consistent result. Once all signals have been resolved, all rules are satisfied and consequently the design complies to the architecture. Only when rules are not being checked, signals might still occur. The entire result is like the team has worked as one architect. Since abstract discussions (e.g. about terminology) are avoided, the team effort as a whole is more manageable and predictable.

The project manager can benefit from the list of signals, because it measures rule violations in an objective way. This provides managers with real-time feedback on progress in the team. It reduces their dependencies on reports from team members, which may be subjectively flawed. Besides, the lists of omissions and ambiguities provide an attractive means for work distribution among team members.

An acquirer gets more assurance about the quality of designs. The absence of signals about a particular
rule means that the design satisfies that requirement for 100%. Besides, more predictable design times translate directly into a reduced project risk. Finally, and most importantly, every business rule satisfied is a business requirement fulfilled. This can even be guaranteed in writing and signed off by a chief architect.

Customers have indirect benefits, albeit not less noticeable. For consistent architecture yields a flexible and maintainable system, which enables the organization to respond adequately and flexibly to the individual and continuously changing needs of their customers.

Besides results for stakeholders, there is one observation of scientific interest. The responses of subjects in the usability experiment have provided a new insight. Apparently, the set of rules coming from ArchiMate were not sufficiently relevant for the architects. They were considered too general. Architects require a more specific level, but this would make ArchiMate either impractically loaded with terms or far too specific to be of use for many architecture projects. This is subject for further research.

Our findings correspond to predicted findings in earlier work [8]. Benefits of concreteness in architecture and a speed-up of the work of an architecture team were corroborated in this experiment.

4. Conclusions

Monitoring architecture processes by means of an automated checker can bring repeatability in innovation. This has been demonstrated by building the checker and performing the usability experiment.

The Archimate reference manual has proven to be an adequate basis for building tools. Practically all of that manual could be implemented directly.

The usability experiment has shown that real-time feedback provided by the checker is definitely an improvement of the architecture process. It allows architects to act more professionally, and renders the architecture process more predictable and reliable.

Further research must be conducted to include project specific rules into the checker.

10. References


Analysis and Verification of an Automated Parking Garage

Aad Mathijssen and A. Johannes Pretorius
OAS Group and VIS Group of Mathematics and Computer Science Department
Technische Universiteit Eindhoven
a.h.j.mathijssen@tue.nl, a.j.pretorius@tue.nl

Abstract

We discuss the software design of an automated parking garage. Our major focus is on safety. For the design of this safety part we have used behavioural modelling techniques. This amounts to creating a high-level behavioural model of the design, and checking if this model satisfies a set of requirements.

The model is created incrementally using simulation, with which we can investigate specific scenarios of the system. A custom made visualisation tool greatly improved the speed and ability with which insights in the design were obtained. More importantly, it made communication of the design to people from outside the field of behavioural modelling techniques much more effective. Finally, by means of verification we have checked the requirements the design should satisfy. This means that each requirement is checked on every state of the model.
Smarter selling of testing

Tim Koomen

VVSS, Eindhoven
November 24, 2005

Why?
What?
Who?
When?
How?
Value of testing

- Finding (and correcting) defects prevents damage
- Known defects still prevent some damage
- Confidence
- Project tracking information

(from Rex Black’s keynote at Eurostar 2002)
When?

- At the start: plan
- During and to the end: progress and results
- Less: at the end...

How...?

- Use a transparent process, using business terms
- Substantiate with facts
- Employ your soft skills
How: test information

- Cost
- Time
- Risks
- Benefits
  - What can we earn
  - What goals will be achieved

At the start

- Assignment
- Product Risk Analysis
- Test Strategy
- Estimation
- Planning + FBL
- Test Activities
- Reporting
- Test Process Management
- Estimation
- Planning + FBL
- Test Activities
- Reporting
- Test Process Management
- Estimation
During and at the end ...

- Assignment
- Product Risk Analysis
- Test Strategy
- Estimation
- Planning + FBL
- Test Process Management
- Test Activities
- Reporting

Defects reports

- Errors
  - Faults/defects
  - Failures
  - Damage

Scope of defect report
**Progress report**

- Test progress
- Quality of the test object (benefits)
  - status (this moment)
  - trends (history)
  - related to project progress
- Product risks
- Bottlenecks
- Quality of test process (optional)

*Advice* on the quality (risks / alternatives)

---

**Soft skills**

- Communication
  - Language of the receiver, WIIFM
  - Advisor, sales
- Presentation and writing skills

- Neutral, objective
- Trustworthy
- Persistent but pragmatic
Summary

www.tmap.net
Towards Pattern-Oriented Test Development based on Abstract Test Notations

Alain-G. Youffo Feudjio
Fraunhofer Fokus, Berlin, Germany
youffo@fokus.fraunhofer.de

Abstract

The Testing and Test Control Notation TTCN-3 [1] is increasingly gaining popularity in testing reactive systems for conformance, interoperability and performance. TTCN-3 is a standardized test notation which resulted from redesigning the Tree and Tabular Combined Notation TTCN. Reaching far beyond TTCN’s traditional domain of protocol conformance testing, TTCN-3’s scope now includes almost all kinds of testing of software-based systems. In the TT-medal project we looked into approaches for enabling and facilitating the reuse of TTCN-3 test artifacts to speed up the TTCN-3 test development process and reduce costs. The use of patterns in the general software development process has proven to be potentially beneficial in helping to achieve those goals. In a previous work, we proposed to introduce that concept to TTCN-3 test development and pointed out which phases of the process would benefit most from it. In this paper, we present how a pattern-oriented test development process can effectively reduce the production time for test systems and more precisely for the specification of the abstract test suite (ATS). Our approach combines automated generation of test skeletons with manual processing. Some examples produced with prototype implementations are also presented to underline the validity of the concept.

1. Introduction

The benefits of using a standardized abstract test notation like TTCN-3 have been described in numerous publications anterior to this one. The fact that, it is a general-purpose and technology independent test notation lead it to be adopted in a wide range of application domains beyond the traditional telecommunications and datacom sectors. However, despite all those advantages and the maturity reached by the language, the process of specifying test suites with TTCN-3 can still be quite tedious and error-prone. This is especially the case for those new application areas originally not covered by its predecessor TTCN-2. One reason for that is the fact that many practical experiences of using the notation for other purposes than protocol conformance testing in big projects have not yet been published. E.g. the TT-medal CORBA-Testing case study in the TT-medal project was actually, to our best knowledge, the first attempt to effectively use the IDL-TTCN3 mapping standard to test real CORBA-based systems. Some of the issues we identified in that case study even go beyond IDL, because they are related to testing of operation- (i.e. synchronous communications-) based systems in the broader sense. The use of test patterns and their integration in the test development process aims at reducing those difficulties by embodying the knowledge acquired while developing test solutions into the process, so that future solutions would be achieved faster and more efficiently.

2. The TTCN-3 Test Development Process

In [2], we presented the TTCN-3 development process which can be decomposed in the following phases:

2.1. Phase 1: Defining the test configuration elements

This phase consists in:
- Identifying the interfaces provided and required by the SUT and modeling them in equivalent Parallel Test Component (PTC) types in the ATS with corresponding ports mapping those interfaces;
- Defining the type of PTCs the test system will use to communicate with the SUT. Those PTCs must provide the ports needed for connecting them with the SUT and optionally some ports for internal communication within the test system, e.g. for synchronization or coordination purpose.

1 www.tt-medal.org
2.2. Phase 2: Defining the type system for the test suite

The type system consists of all the data types describing message or data structures required to communicate with the SUT. TTCN-3 supports the import of types and values defined in other languages e.g. ASN.1, IDL, XML, etc. Therefore, this phase of the development process might be achieved automatically, using appropriate tools.

2.3. Phase 3: Specifying the data required for testing

In TTCN-3, so-called templates are used to define data transmitted to or received from the System Under Test (SUT). TTCN-3 templates can also be used to describe the parameters of a method provided or used by the SUT via one of its interfaces. Those templates are generally referred to as “signature templates”.

2.4. Phase 4: Describing the test behavior

To express test behavior, TTCN-3 supports all the features common to functional programming languages such as loops (for-, while-), functions, if-statements etc. plus some concepts specific to testing; e.g. test cases, test steps, and matching mechanisms for evaluating SUT reactions.

3. The Pattern-oriented TTCN-3 Test Development Process

3.1. Overview

Pattern-oriented test development consists in integrating concepts of recurring solutions in the test development process. In [2], we identified three main categories of TTCN-3 test patterns:

- Architectural patterns describe how test components can be composed and connected to a SUT to test it for conformance, performance or interoperability.
- Data patterns describe approaches for specifying TTCN-3 test data.
- Behavioral patterns encapsulate the knowledge gathered in defining test behavior with TTCN-3.

As depicted on Figure 1 below, each of these categories of patterns can be used in the TTCN-3 test development process to generate parts of the abstract test suite automatically and therefore, fasten the process.

3.2. Architectural patterns in the TTCN-3 test development process: Automated Generation of Test Configurations

In situations where the SUT has been specified using IDL, UML component diagrams or any other notation for which a mapping to TTCN-3 can be defined, automatic generation of test configurations can be achieved. Depending on whether component-level or unit-level testing is targeted, basic test configuration elements such as component types, ports and timer variables could be generated automatically to build the test system. For example, if the SUT can be represented as a component which provides \( n_p \) interfaces and requires \( n_r \) interfaces, then any test system for that SUT could be composed, based on one TTCN-3 parallel test component type providing \( n_r + n_p \) ports and one main test component type for coordination purposes. This is illustrated on Figure 2, which features component or system-level testing of an SUT providing two interfaces (One operation-based interface and one message-based interface) and requiring one (message-based) interface. As depicted on that picture, the same parallel component type can be used to build two
different test configurations for the SUT. Furthermore, a test configuration for load/performance testing could also be achieved using the same approach. To obtain the test configuration, we apply the following patterns:

For each interface provided or required by the SUT:
- Define two different port types to represent the interface type in the test system. One of them could be used for testing outgoing communications (synchronous or asynchronous) to the SUT, while the other one’s purpose is to handle incoming communications from the SUT e.g. replying to incoming messages or synchronous requests from the SUT via that interface.
- Define a parallel test component type aiming at testing the functionality provided or required by the SUT via that interface.
  - The defined component type has at least one instance of the different port types mentioned above to be able to support duplex communication with the SUT according to the rules described above.
  - Define a timer variable in the component type to be used in behaviours involving the test component for deadlock avoidance. Timers are essential in test systems to avoid deadlocks in testing reactive systems. E.g. if a timer is not started before a stimulus is sent to the SUT in expectation of a response and for any reason whatsoever, the SUT does not respond according to the specified expectations, then the test systems enters in a deadlock state and the test will have to be interrupted without any verdict. The value of the generated timer should be set to a default value representing the maximum delay to be expected when issuing requests or messages to the SUT. This default value must be customizable via the management interface for more flexibility.
- Define a component type representing the functionality provided or required by each interface of the SUT that is externally visible. This component type has the same ports as the one mentioned above with the only difference being that, in this case no timer definition is required.

The test configuration we define with this pattern is suitable for unit-level testing, but could not be used for system-level testing which generally involves several different interfaces. For subsystem and system level testing we apply the following pattern to obtain the test configuration:
- Define a component type containing ports representing all interfaces available at the SUT following similar rules as those mentioned above. The difference lies in the fact that, this time around the defined component type has ports allowing it to support bi-directional communication with all interfaces provided or required by the SUT.
- In analogy to the previous rule, define a component type representing the whole SUT and containing ports mapping all its interfaces to allow mapping operations in the ATS.

3.3. Data Patterns in the TTCN-3 Development Process: Generation of Test Data

The specification of test data is the most time-consuming part of TTCN-3 test development. This fact becomes more obvious for systems in which complex structured data types are used containing several dozens of fields, with some of those also of complex structured types. Specifying templates to represent test data for those data types is then a highly error-prone and thus time-consuming activity, if appropriate tool support is not available. Currently no TTCN-3 test specification environment provides tool support for template definition in the form of context-sensitive type completion, wizards or skeletons. Therefore improving this process will have a deep impact on the test development process as a whole in terms of production time and costs reduction. With semi-automatic generation of test data, we can dramatically fasten TTCN-3 test data specification. The approach consists in generating TTCN-3 data patterns, i.e. reusable generic TTCN-3 templates and parameterized templates automatically, that can easily be imported and reused as-is by the test developer, or customized with little effort using TTCN-3’s modifies keyword. The pattern used for generating the test data is as follows:

For each structured data type potentially exchanged as a message or a parameter in the communication between test system and SUT, define a generic template for outgoing communication from the test system to the SUT. For such templates the following rules are used:
- The value for all optional fields is set to omit.
The value for simple type fields is set to a default value based on a module parameter whose exact value could be modified by the test executor through the test management interface.

The value for structured type fields is set to a generic template of the processed field’s type. Enumerations and Unions must be treated differently, because their value depends on the actually selected variant. One possible approach for solving this issue is by providing a facility for the test developer (i.e. the person writing the test suite) to indicate which of the variant should be selected per default and then use that variant every time a value of the enum- respectively union type is needed. Another approach might consist in generating a different template for each of the possible variants of a union or enum type. However, this might lead in some cases to an explosion of the number of possible combination and hence too much code being generated with the potential of breaking existing tools by exceeding the maximal supported file size. For that reason, we opted for the first solution and in case that a variant was not selected as default, we assume that it does not matter for testing and chose one randomly. Figure 3 below depicts an example of TTCN-3 data type specification copied from a test suite for the SIP protocol and Figure 4 contains the generic TTCN-3 template generated for the L_Message_Request type depicted on Figure 3.

Furthermore another generic template for incoming communication at the test system from the SUT is generated using the following rules:
- The value for all optional fields is set to “*”
- The value for any non-optional field is set to “?”
- If a field is of record or set type and all its subfields are optional, then that field is set to “?”

Figure 3. Example of TTCN-3 data type definitions

Figure 4. Example of automatically generated outgoing message template

Figure 5 below presents an example of generic incoming template, based on the same type as the outgoing template depicted on Figure 3.
The generic templates can also be reused to define new data using the TTCN-3 modifies keyword. This is illustrated by the code snippet on Figure 6, which features reuse of the template definition displayed on Figure 4.

```
template L_MESSAGE_Request
L_MESSAGE_Request_s_0 := {
    requestLine := {
    msgHeader := ?,
    messageBody := *
}
```

**Figure 6. Example of reuse of generic template (cf. Figure 3)**

### 3.4. Behavioral Patterns in the TTCN-3 Development Process: Generation of Test Behavior

A test behavior pattern can be defined as a (s,r,P) triple, i.e. the combination of a stimulus s, the response r the test system (TS) expects or initiates following that stimulus s, given a set of parameters or constraints P.

Depending on whether the SUT uses a synchronous or an asynchronous communication scheme different behavior patterns can be used to generate TTCN-3 test skeletons that will provide the base for specifying more complex test scenarios, i.e. sets of (s,r,P) triples in sequence or running in parallel.

**Table 1** below lists the behavior patterns that are applicable in the case of a synchronous communication scheme, for each method m (i.e. equivalent to a corresponding TTCN-3 signature) available at the SUT and potentially raising $n_E$ types of exceptions.

```
<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Response</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS issues a call of method m to the SUT</td>
<td>Method m returns normally: returned value is irrelevant</td>
<td>- Returned value (irrelevant, user-defined) - Returned Parameters (irrelevant, user-defined)</td>
</tr>
<tr>
<td>Incoming call from SUT</td>
<td>TS returns normally</td>
<td>- Returned value (user defined, default) - Returned parameters</td>
</tr>
<tr>
<td>TS raises exception of type E</td>
<td>- Exception type and value</td>
<td></td>
</tr>
</tbody>
</table>
```

**Table 2** Behavior patterns for SUTs supporting asynchronous (message-based) communication

For SUT supporting asynchronous (message-based) communication, the behaviour patterns can be more complex, because the sequence of events is less predictable. However, as displayed in Table 2 below, a set of behaviour patterns can also be identified for that case and used in the test development process to optimize it.

```
<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS sends message A to the SUT</td>
<td>SUT sends message B</td>
</tr>
<tr>
<td></td>
<td>SUT discards message A =&gt; Time out at TS</td>
</tr>
<tr>
<td>TS receives message A from SUT</td>
<td>SUT sends message sequence B, C and D</td>
</tr>
<tr>
<td></td>
<td>SUT sends $n_r$ retransmissions of B</td>
</tr>
<tr>
<td></td>
<td>SUT sends one of message B, C or D</td>
</tr>
</tbody>
</table>
```

**Table 2** Behavior patterns for SUTs supporting message-based (asynchronous) communication
If the test system’s configuration is available, along with the data types of the messages or parameters to be exchanged between the test system and the SUT, then the patterns listed in Table 1 and Table 2 can be used to generate elements of TTCN-3 test behaviour automatically. To illustrate the approach, we introduce the following example of an SUT supporting operation-based communication as depicted on Figure 7 below. Figure 7 depicts the representation of an SUT providing one operation-based interface consisting of 4 methods in TTCN-3. Such a representation could be generated automatically from the SUT’s specification language (IDL, WSDL etc.) using a translation tool.

```tcl
group MyInterfaceInterface__ETSI { 
signature myMethod1(in float in_p, out float out_p); 
signature myMethod2(in float in_p, 
inout float in_p_2); 
signature myMethod3(in MyRecordType rcd_in, out MyUnionType union_out); 
signature myMethod4( 
in MyUnionType union_in_p, 
out float out_p) return MyRecordType 
exception { 
MyExceptionType}; 

type port MyInterface procedure{ 
inout myMethod1; 
inout myMethod2; 
inout myMethod3; 
inout myMethod4 
} 

type address MyInterfaceObject; }
```

**Figure 7. TTCN-3 Representation of an SUT**

Applying the first pattern listed on Table 1 for generating reusable code snippets of test behaviour for the SUT defined on Figure 1 lead us to the following result:

For each signature present at the SUT’s interfaces 2 signature templates are generated, with one for outgoing requests on that signature and the other one for incoming requests. Figure 8 below presents an example of signature templates generated from the SUT’s specification depicted on Figure 7.

```tcl
template myMethod4 myMethod4_s_0 := { 
union_in_p := { 
setVar := { 
float_field := DEFAULT_FLOAT, 
oct_field := DEFAULT_OCTETSTRING, 
hex_field := DEFAULT_HEXSTRING }}, 
out_p := - }

template myMethod4 myMethod4_r_0 := { 
union_in_p := -, 
out_p := ? }
```

**Figure 8. Example of automatically generated signature templates**

For each interface provided or required at the SUT a set of helper functions is generated for client-side and server-side testing of the SUT. Client-side testing means that the SUT uses the interface and that the test system acts as a component providing that interface as a service. On the other hand, server-side testing means the SUT provides the interface as a service and that the test system acts as a client to that service.

For each signature of a given interface, a function encapsulating a call of that signature is generated, which takes into account the fact that the signature might return a value or throw a previously defined exception. The generated signature should not be coupled to any configuration, but take the port to be used as parameter to facilitate reuse in another context.
4. Conclusions and Outlook

We applied our approach of pattern-oriented TTCN-3 test development to implement a conformance test suite for the OSA-Parlay API with great success. The specified test suite was based on the test suite structure and test purposes document proposed by the ETSI for conformance testing of OSA-Parlay implementations. We could generate more than 80% of the required test code automatically. This was a clear indication that the use of patterns in the TTCN-3 test development process bears great potential, especially for systems using synchronous communication.

We are in the process of further investigating approaches for pattern-based test development for systems using asynchronous communication.

Furthermore, we believe that the introduction of a meta-language for testing, that would focus on the test intent and the test scenario and hence would combine the strength of a standard test notation like TTCN-3 with the more abstract concept of test patterns, would be very beneficial for test- and system developers alike. However, to ensure that we do not create yet another (test) notation, analyzing existing notations such as UML or the UML2 Test Profile (U2TP) will on suitability for that purpose will be a prerequisite of any further work in that direction.

5. References


Figure 9. Example of TTCN-3 help functions for SUTs supporting synchronous communication
WHAT’S NEW IN
MERCURY BUSINESS PROCESS TESTING 8.2.1:
THE “NEXT WAVE” OF FUNCTIONAL TESTING GETS EVEN BETTER
INTRODUCTION

Quality assurance (QA) engineers are responsible for assuring the viability and functionality of the enterprise’s mission-critical applications. But applications turn out better when the enterprise’s line of business (LOB) experts help support QA’s vital role. Limited business analyst involvement during testing can lead to miscommunications and defects and breakdowns in critical business processes. Conflicting priorities between content experts and quality engineers result in time-consuming test rework.
To ensure the health of an enterprise’s applications, testing should be conducted throughout the application lifecycle. Defects caught early in development are much easier and less expensive to fix than problems uncovered late in the lifecycle or in production. One key to developing and launching high-quality applications is to involve business analysts early in QA’s application testing processes. Input from these content experts can help QA determine if the applications are meeting all business requirements and better ensure the proper functionality is being developed correctly and thoroughly tested.

The classic problem with involving business analysts early in the testing cycle is that most of today’s functional testing products are too technical for anyone other than highly skilled quality engineers to use. This technical hurdle has now been solved with the introduction of easy-to-use business process testing solutions. These solutions enable content experts who know how the applications are supposed to work to play a supporting role in QA. With intuitive tools, these individuals can easily write tests based on what application functionality they need. Involving business process experts early in the quality lifecycle complements QA’s testing processes, enhancing the quality and functionality of the enterprise’s key applications.

**Mercury Business Process Testing Overview**

Mercury Business Process Testing™ provides a complete role-based test automation system that enables content experts to build, data-drive, execute, and document test automation without any programming knowledge, allowing them to focus on creating high-level test flows that mirror actual business process. This contribution to QA’s testing efforts can free up more technical QA engineers to concentrate their efforts on areas that facilitate automation.

Mercury Business Process Testing does the following:

- Greatly simplifies and speeds up the test design process by using reusable components (business process building blocks).

- Allows QA and testing teams to start the test design process much sooner – during system design – accelerating time-to-deployment for high-quality software.

- Generates automated tests and test case documentation in a single step, eliminating the expensive and time-consuming processes of creating and maintaining test records.

- Enables QA teams to use prepackaged test assets and best practices to implement test automation for leading enterprise resource planning (ERP) and customer relationship management (CRM) applications, saving time and leveraging the knowledge of experts.

- Eases the adoption of test automation, because the solution is so easy to deploy and use.
Mercury Business Process Testing also enables enterprises to leverage their investments in the tools they have already purchased. It is part of Mercury Quality Center™, an integrated set of software, services, and best practices for automating key quality activities, including requirements management, test management, defect management, and functional testing. Mercury Business Process Testing integrates smoothly with any work already done with Mercury QuickTest Professional™ or Mercury WinRunner® – Mercury Quality Center products that support more than 90 percent of the Fortune 500 and more than 65 percent of all automated software quality initiatives.

Mercury Business Process Testing allows for significant increases in the productivity of subject matter experts and QA/test engineers alike. As such, many IT organizations have seen measurable return on investment (ROI) benefits and fast payback from their investments in Mercury Business Process Testing.

**New Features in Mercury Business Process Testing 8.2.1**

Mercury Business Process Testing version 8.2.1 offers several significant new capabilities. The following sections will describe some of the new product enhancements:

**1. Support for Mercury WinRunner Customers**

Mercury WinRunner is one of the most widely used functional and regression testing tools in the industry. For many years, customers have been creating tests assets in WinRunner to support their QA initiatives. Business Process Testing 8.2.1 now supports Mercury’s existing WinRunner customers. Mercury WinRunner users can now leverage integration with Mercury Business Process Testing to accomplish the following:

- Plug into the industry’s only web-based, end-to-end collaborative platform for scaling quality automation.
- Test sooner in the software lifecycle, even before the application is delivered to QA.

Mercury WinRunner users can leverage integration with Mercury Business Process Testing to:

- Convert existing programmatic scripts into Mercury Business Process Testing components.
- Create new scripted components in Mercury WinRunner 8.2.
- Combine Mercury WinRunner and Mercury QuickTest Professional components together in a Mercury Business Process Testing test.

Mercury WinRunner customers can either create new Mercury Business Process Testing components directly using WinRunner, or convert their existing WinRunner scripts into reusable Mercury Business Process Testing components. Tests created in WinRunner can be edited and debugged within WinRunner using the same processes that are familiar to users who work with WinRunner tests today.
In the past, Mercury customers had to choose whether to build their test assets in either Mercury WinRunner or Mercury QuickTest Professional, based on the particular application environment they were testing. With Mercury Business Process Testing 8.2.1, that requirement becomes irrelevant. Now customers can create end-to-end test scenarios that cover environments as diverse as mainframe and .Net using a single, unified solution.

One of the most significant benefits gained from Mercury Business Process Testing support for Mercury WinRunner is that Mercury Business Process Testing automates what the majority of WinRunner customers are already doing today by using complex Excel spreadsheets and text files. (To take advantage of the many benefits offered by Mercury Business Process Testing, Mercury WinRunner customers will need WinRunner version 8.2 and must be running Mercury Quality Center and Mercury Business Process Testing 8.2.1.)

2. User Acceptance Testing

User Acceptance Testing is the last phase in the QA process when LOB users certify and sign-off on test plans and tests. The need for User Acceptance Testing is becoming even more critical as more testing projects are outsourced and off-shored. It is the only way the business can validate the work done by third-party testing teams.

With support for User Acceptance Testing, Mercury Business Process Testing 8.2.1 makes it easy for quality automation teams to close the loop with the business. Testing teams can run existing business process tests manually in the Test Lab module. Each component iteration is treated as a step in the test. Testers can view and use input and output parameters in the steps and can store the results of each component in the manual test run without having to duplicate any additional work in Microsoft Word or Excel.

3. Component Grid View

With Mercury Business Process Testing 8.2.1, it is now possible to view all components in a project in the grid view, which offers advanced search and filtering capabilities.

4. Copy/Paste Support

Mercury Business Process Testing 8.2.1 provides the ability to copy and paste components, business process tests, and test sets containing business process tests within and between Mercury Quality Center projects and servers.
5. New Component Options in the Document Generator

It is now possible to include component information for all or selected components in project documents that are created using the Document Generator. The component information can include component step details, attached snapshots, and the list of tests that use each component.

6. Enhanced Component Request Wizard

When creating component requests, a new step in the wizard provides the ability to enter manual steps for the components.

7. Non-Automated Components

New components in the Business Components module are created as non-automated components. Testers can add manual steps to the component and run the component manually within a business process test. It is also possible to convert a non-automated component to an automated Mercury WinRunner or Mercury QuickTest Professional component. When converting a non-automated component to an automated component, any existing manual steps are converted to comments within the automated component.

New Accelerators

One of the most exciting realities of Mercury Business Process Testing is that it has enabled an ecosystem of partners to build value added solutions—called Accelerators—that run on top of the Business Process Testing Platform. Business Process Testing Accelerators are pre-packaged customizable business components and test flows that significantly reduce time-to-test.

The Accelerator concept is simple. Customers can deploy pre-packaged test solutions faster than if they were build them on their own. And because Accelerators are built using the Business Process Testing Platform, they are cheaper, easier, and require less work to maintain and upgrade than traditional test scripts.

Mercury recently teamed up with solution partners who specialize in ERP/CRM and technology vendors in the security testing space to deliver additional Mercury Business Process Testing Accelerators for SAP, Oracle, and security testing.

Implementing a Complete End-to-End Solution

When adopting any new technology, organizations must focus on managing change so that it happens quickly and with the least amount of disruption. This is why Mercury offers comprehensive consulting services, to make it easy for organizations to get up and running with Mercury Business Process Testing. Mercury Consulting™ uses a proven implementation methodology to guide customers through their complete project lifecycle. As part of this approach, Mercury includes a review of the changes involved in the customer’s day-to-day processes, as well as guidance on user adoption and overall rollout of Mercury Business Process Testing. Mercury best practices are used as key part of the services delivery. These include methods through which content experts can assist in the testing process. Mercury Consulting Services also provides documentation describing the products, people, and process best practices for Mercury implementations.
To help ensure that customers are successful rolling out and scaling Mercury Business Process Testing throughout their organizations, Mercury Consulting Services for Mercury Quality Center offers two service delivery options: Mercury Business Process Testing QuickStart™, and time and materials engagements that are mapped out to meet customer’s specific needs. Both options allow Mercury Consulting Services to help customers leverage Mercury best practices, maximize ROI, and ensure the lowest possible risk to the customer’s Mercury Business Process Testing initiatives.

Both of these services offer excellent implementation guidance and training to maximize Mercury Business Process Testing’s role-based team collaboration features and quickly make customers self-sufficient on the solution.

Summary and For More Information

Business process testing solutions close the gap between the business needs of the application and the enterprise’s more comprehensive QA testing processes. Close collaboration between the enterprise’s business process experts and QA team makes testing processes much more efficient and results in higher-quality applications.

Mercury Business Process Testing removes the technical complexity and specialized expertise from the test design process. Subject matter experts can facilitate early testing by focusing on business processes rather than running the tests. It also centralizes and simplifies test and documentation creation and maintenance, resulting in substantial savings for today’s enterprises.

Mercury’s newest release – Mercury Business Process Testing 8.2.1 – brings even more features and functionality to the testing process. With support for Mercury WinRunner customers, User Acceptance testing, and more, Business Process Testing delivers an even higher level of productivity to test teams and dramatically improve the quality of finished software applications. For more information on Mercury Business Process Testing or any Mercury products and services, please visit www.mercury.com.
Six Sigma
And The
Compuware
Application
Reliability
Solution

Compuware Corporation
One Campus Martius
Detroit, Michigan  48226
313.227.7300

Scott Margolis, PMP
QA/PM Subject Matter Expert

Salil Raje
Six Sigma Black Belt

August 2004
**Background**

In the late 1980’s, Motorola developed a business process to continuously improve manufacturing processes. Through a process of defining what is to be measured, measuring the results of the process, analyzing the results, implementing improvements and changes to improve the process, and controlling the overall process, an organization can work towards a manufacturing goal of 99.97% accuracy, 4 defects per million, six standard deviations or “Six Sigma” of a normal statistical distribution.

This document is being written to provide insight into what Six Sigma is, and how a CARS implementation relates to those organizations considering or embracing Six Sigma. Six Sigma at its basis is a manufacturing process that to this point, has related primarily to objects that can be physically measured, such as with a micrometer or other physical measuring device, or with integer values for those items found to be acceptable by other more subjective measures. The further purpose is to provide better service to our customers who may be interested in or are in the process of organizationally adopting the Six Sigma approach and may wish to learn how a CARS engagement relates.

**Compare and Contrast:** The broad assumption in this document is that the reader has been previously exposed to the Compuware Corporation CARS offering to some extent. Some of the terminology utilized within this document may be specific to CARS, and some to Six Sigma practices. The objective is to compare similar ideas within the two methods to provide a baseline from which to understand the use of CARS within an organization in process of, or considering the adoption of, Six Sigma processes.

**What Is Six Sigma?**

Six Sigma is a multi-faceted approach to business improvement. It includes a philosophy, set of metrics, set of improvement frameworks and a toolkit. When discussing Six Sigma, it is important to put in context to which of these aspects we are relating.

**Six Sigma as a philosophy:** The Six-Sigma philosophy is to improve customer satisfaction through defect elimination and prevention and as a result, to increase business profitability. “Defects” are defined in terms of the customer’s (not engineer’s) point-of-view. Bear in mind that a customer’s view may be either (or both) internal or external. The business profitability motive is crucial; improvement for improvement’s sake, without positive impact on the bottom-line, does not align with the Six Sigma philosophy. Six Sigma was originally targeted at manufacturing operations and, due to the phenomenal success of Six Sigma in this environment, has lead to a
Six Sigma projects begin and end with business considerations. Project selection and tracking focus on maximizing the benefit delivered to the business bottom line. While there may be plenty of fundamental metrics and statistics en route, Six Sigma project success is measured in financial terms. ‘Process maturity” is not an interest in itself – the focus is on quantitatively measured business benefits. Perhaps the most important distinction between Six Sigma and other approaches to process improvement in software lies in its almost obsessive preoccupation with financially measured business results. Six Sigma caters primarily to the concerns of the CEO and CFO – process maturity is not viewed as a business benefit in and of itself. Those organizations adopting CARS and the QualityPoint™ method have found process maturity comes as a beneficial by-product.

Success of Six Sigma in software requires more than just an understanding of the Six Sigma philosophy and tools. It also requires learning how the tools and philosophy apply to the specific business area being addressed.

**Six Sigma frameworks** - There are currently two main Six Sigma frameworks: DMAIC and DFSS.

**DMAIC** (Define-Measure-Analyze-Improve-Control) is used to improve and optimize existing processes and products. This may be heard pronounced “duh-may-ick” within Six Sigma conversations.

**DFSS** (Design for Six Sigma) is used to design new products and processes. It is also used to redesign existing processes and products that have been optimized but still do not meet performance goals. DFSS uses DMADV (Define, Measure, Analyze, Design, and Verify) as steps.

When thinking about the connection between Six Sigma DFSS/DMADV and DMAIC one can visualize a temporal relationship and a tendency for these views to live in different quadrants of the Six Sigma space. The relationship is temporal in the sense that one clearly cannot apply DMAIC to a product or process that does not exist (i.e. software), so in that sense DFSS comes first—although clearly many products and processes exist that were not created using the DFSS approach. Hence, the boundary between DFSS and DMAIC is “fuzzy” in practice. When products or processes were created using DFSS we will have created a lot of valuable information and context that can be revisited to advantage when we later start a DMAIC project. When that is not the case, we may
need to reach back into the DFSS space from within a DMAIC project to create what is missing.

The boundary is also fuzzy in the sense that DFSS tends to focus externally and strategically, while DMAIC has a tendency to focus internally and tactically. Broadly speaking, DFSS projects are often more closely connected to the voice of the customer (VOC), while DMAIC projects are often more closely tied to the voice of the business—as with every generalization, there are exceptions and border conditions.

**Six Sigma metrics** – 3.4 defects per million opportunities is the most cited metric. Other measures are defect rate (parts per million), Sigma level, Defects Per Unit (DPU), and Yield.

Sigma is a Greek letter used to describe the amount of deviation in a process or procedure. In the parlance of the statistician, sigma is the term applied to one standard deviation from the mean of a population (μ) or sample (σ). An inclusive, higher sigma value indicates less deviation or fewer defects. The central idea behind Six Sigma is that if you can measure how many “defects” you have in a process, you can systematically figure out how to eliminate them as close to their source as possible and get close to “zero defects”. This same philosophy is embodied in the CARS QualityPoint™ method.

**Six Sigma toolset** – relate to the 5-steps of the DMAIC process as per the following:

<table>
<thead>
<tr>
<th>Define</th>
<th>Measure</th>
<th>Analyze</th>
<th>Improve</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>♦ 7 basic tools -</td>
<td>♦ Cause and Effect Diagrams</td>
<td>♦ Robust Design</td>
<td>♦ Non-Statistical Controls: ♦ Procedural adherence ♦ Performance Management ♦ Preventive Activities</td>
</tr>
<tr>
<td>Baseline</td>
<td>♦ Defect Metrics</td>
<td>♦ FMEA</td>
<td>♦ Tolerancing</td>
<td></td>
</tr>
<tr>
<td>Project Charter</td>
<td>♦ Data collections methods</td>
<td>♦ Decision and Risk Analysis</td>
<td>♦ Modeling</td>
<td>♦ Statistical Controls: ♦ Control Charts ♦ Time Series Methods</td>
</tr>
<tr>
<td>Kano Model</td>
<td>♦ Sampling Techniques</td>
<td>♦ Capability</td>
<td>♦ Design of Experiments</td>
<td></td>
</tr>
<tr>
<td>Voice of the Business</td>
<td>♦ Measurement System Evaluation</td>
<td>♦ Reliability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voice of the Customer</td>
<td>♦ Systems Thinking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QFD</td>
<td>♦ Root Cause Analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process Flow Map</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Management by Fact</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note: It is important to remember that the Six Sigma toolkit is dynamic and organization-specific. The decisions to adapt, add, or focus on specific methods should be based on the improved ability to deliver on customer needs and business benefit.

**QualityPoint™**

The process used within the CARS solution to drive applications toward higher quality is the Compuware Corporation patented QualityPoint™ method.

**Where’s the risk?** Among the processes followed within Six Sigma is the early determination of areas or items of risk. By the use of a process of risk identification and quantification, areas of exposure in a manufacturing or other process can be ascertained early. As a result, those areas with the potential to cause the most problem can be planned for and risk management strategies instituted. One of the process tools that can be applied to this risk identification process in Six Sigma is called the Failure Mode & Effects Analysis, or FMEA (fuh-me-uh). Key items related to the cause and effect, frequency of occurrence, the “detectibility” of defects and possible costs of defects (value) are inserted into the model. The result is a detailed listing of what can go wrong in a manufacturing system or process, with a prioritization (Risk Priority Number) listing allowing organizational management to accept, mitigate or transfer risk as is most economically prudent, as well as recalculate the risk score after a risk strategy is selected. Figure 1 shows what a FMEA for a process might look like:

<table>
<thead>
<tr>
<th>Service/Process</th>
<th>Potential Failure Mode</th>
<th>Potential Effects of Failure</th>
<th>SEV</th>
<th>OCC</th>
<th>DET</th>
<th>RPN</th>
<th>Recommended Action</th>
<th>Who Acts</th>
<th>Action Taken</th>
<th>SEV</th>
<th>OCC</th>
<th>DET</th>
<th>RPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter a Order</td>
<td>Order is wrong</td>
<td>Ordered items need to be returned</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>192</td>
<td>Retrain Order Takers</td>
<td>Sales Mgr</td>
<td>Order-takers retrained</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>On-site recruiting</td>
<td>On-site recruiting process is not implemented</td>
<td>Insufficient number of employees</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>144</td>
<td>Cross-train all recruiters on the on-site recruiting process</td>
<td>Branch Managers 1/15/04</td>
<td>Recruiters cross-trained</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>48</td>
</tr>
</tbody>
</table>

**Figure 1:** A Sample Failure Mode & Effects Analysis

While the FMEA has proven to work quite well for a system of processes such as manufacturing or business processes, its use for software development has not been successfully demonstrated in this format.

The CARS QualityPoint™ method has taken giant strides in remediating the problem of proactive risk determination through the use of the Functional Decision Tree (FDT) and the Test Decision Tree (TDT) that are at the heart of the patented QualityPoint™ method.
Within the process of recording the function points and test cases of the software under consideration, QualityPoint™ and CARS allows organizations to apply risk determination in much the same manner as a FMEA, but with the ability to account for specific risks, or values that are important to the customer and the business (VOC and VOB). The risks can be unique to a project, cycle or line of business. In any case, CARS give the organization the flexibility to determine the factors that are most important to their customer and business situation.

By following the QualityPoint™ risk based testing methods that are an intricate part of the CARS solution, organizations successfully incorporate risk determination and weighting into the process. When completed with this patented process, the distribution of risks appears to approximate a normal distribution. (Figure 2)

Figure 2:
A Normal Distribution of Requirement Priorities

Requirements with Lower Risk
Requirements with Higher Risk

Compuware’s risk management assessment within the QualityPoint™ method is the most effective application of risk evaluation in a process that is specifically designed for use in software systems development. Much like the FMEA, the QualityPoint™ Functional and Test Decision Tree’s help an organization that is either creating new software, or implementing packaged software requiring customization, such as an ERP, CRM or MRP package, to be able to identify early and accurately, those requirements and test cases with the highest risk and the highest value, so that management may take appropriate prioritization and risk mitigation steps in a well planned, well thought out process that leaves nothing to chance. The ability to then improve the process if and when defects are discovered is the distinction between the high degree of flexibility offered by QualityPoint™, and other more rigid software development and testing methodologies.
Six Sigma Elements In CARS

In mapping CARS to the Six Sigma philosophy, we find that CARS is motivated by similar aspects in its philosophy, which is to improve customer satisfaction through defect elimination and prevention and, as a result, to increase business profitability in the context of software and business systems quality. Specifically, CARS addresses the cost of planning for quality, testing software applications, establishing metrics, (Figure 3) and reducing the time it takes to test applications consistently and rigorously. CARS strives to improve Customer Satisfaction at two levels – the users (Voice of the Customer), and the IT Management responsible for delivering quality applications to those users (Voice of the Business). Using QualityPoint™, CARS seeks to prevent defects (as defined by the user) through a focus on Requirements Definition as implemented through the Function Decision Tree and structured use of Compuware integrated technologies. Through the Scope Analysis, Statement of Work and Assessment activities, **CARS seeks to prevent defects (as defined by the IT and QA Management) prior to their emergence as a defect that is recognized in production – the essence of Six Sigma.**

![Figure 3: Establishment of Metrics: Exit Criteria by Requirement Risk](image)

As much as Six Sigma is process-centric, CARS also has a well-outlined delivery process defined by 7 Key Process Area’s (KPA) that account for all quality activities in the software development lifecycle, from planning through process feedback. More importantly the seven KPA’s of QualityPoint™ confirms the "process-centricity" of CARS. In the customization and deployment of CARS, these seven KPA’s are evaluated by the CARS QA Architect against the existing testing processes of the client to determine gaps, which need to be filled to improve the client’s test processes.

At a high-level, the DMAIC steps may be thought to map to the CARS delivery process as per below:
Six Sigma and the Compuware Application Reliability Solution

<table>
<thead>
<tr>
<th>Six Sigma phases or steps</th>
<th>DEFINE</th>
<th>MEASURE</th>
<th>ANALYZE</th>
<th>IMPROVE</th>
<th>CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARS steps</td>
<td>Scope Analysis / Statement of Work</td>
<td>Assessment</td>
<td>Implementation / Delivery and Turn-over</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activities</td>
<td>The Define phase or the Scope Analysis phase concentrates on “defining” the scope of work through dialog with the Project Champion/Sponsor. A Statement of Work is developed.</td>
<td>In the Measure and Analyze phases the CARS Architect assesses the current Test Process in the context of the seven KPA’s, in addition to conducting analysis relating to Goals, Culture, Organization, Measures, Practices, Test Personnel and other areas determined during the Define phase. A gap analysis is conducted and presented to the Project Champion.</td>
<td>During these phases the customized CARS solution is implemented. The QA Architect leverages the current strengths of the client, designs the AQW workflow and templates, using the knowledge gained from the Assessment and the Scope Analysis phases. The trained CARS Core Delivery Team completes the “improvement” objective and ensures knowledge transfer (if needed).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Similarities</td>
<td>CARS Statement of Work has the similar elements as in the Project Charter of a Six Sigma project.</td>
<td>CARS Assessment is similar to Decision and Risk Analysis conducted during a Six Sigma project</td>
<td>The activities involved with the above phases are similar to the concept of Robust Design and Non-Statistical Controls stated under the Six Sigma Toolset above.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Differences</td>
<td>No formal methods of measuring Capability are applied in CARS. See CMM/CMMI</td>
<td>Design of Experiments, Tolerancing, Modeling and Statistical Controls are not applicable to CARS.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One of the major benefits of the CARS process solution might be the use of data to drive process improvement decisions, by Six Sigma projects. CARS is a solution, which is based on industry best practices and so the need for data may not be applicable to a specific project. CARS, however, does attempt to uncover data during the Scope Analysis and Assessment phases to provide for the customized solution, as relevant to the client organization. In this manner, this CARS phase is analogous to Design For Six Sigma.

**Process Drives Technology:** It has been demonstrated in any number of software shops that putting technology in place without a process simply allows organizations to automate bad habits. Much like DFSS, the CARS solution focuses on developing a process for the organization and bringing in the technology required to support the process.

**Benefits Of CARS To A Six Sigma Organization**

Besides the obvious and already stated benefits of CARS, the QualityPoint™/AQW foundation is a desirable prerequisite for application of Six Sigma for Software - a consistent process is necessary for learning and improvement. It is axiomatic: **An organization that has no process, has no process to improve.**
References:


Measurements for Controlling Test Effort and Depth

VVSS 2005
Henry Peters

Introduction

Test Process & Control Problems
Test Process Variables & Metrics
Using Test Metrics, Examples, Results
Test Process

Test Control Problems
- Estimation and Planning
- Adapt to Circumstances
- Determine Stop Moment

Balancing required vs. possible testing

> Metrics
Basic Test Process Variables:
A. System Volume
B. Number of Test Cases
C. Number of Defects

Test Process Variables & Metrics

A. Volume (& complexity)
- Function Point Analysis (FPA)
- New / Modified / Unchanged ?

> Number of Function Points
B. Test Cases
- What is a Test Case?
- Logical = Physical

> Number of Test Cases / Function Point

C. Defects
- What is a Defect?
- Defect Discovery Moment

> Number of Defects / Function Point
> Number of Defects / Test Case

> Number of Defects / Time (testing day)
> Number of Defects found/not found
Effort Estimation:

Effort = Function Points * \( \frac{\text{Test Hrs}}{\text{FP}} \)

\[ \text{Number of Test Cases / FP} \times \frac{\text{Hrs}}{\text{Test Case}} \]

Test Depth  Productivity

Number of Test Cases / FP:
Indications, from Implementation Aftercare:

<table>
<thead>
<tr>
<th>test cases / fp</th>
<th>test quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>minimal</td>
</tr>
<tr>
<td>0.5</td>
<td>moderate</td>
</tr>
<tr>
<td>1</td>
<td>reasonable</td>
</tr>
<tr>
<td>1.5</td>
<td>good</td>
</tr>
</tbody>
</table>
**Function Points and Test Cases?**

Simply entry function:
- 1-15 DETs, 1 LGV
- 1-4 DETs, 2 LGVs

= 3+ function points

Test with 3+ Test Cases?

---

**Test Quality Elaboration: the Detection Rate**

\[
\text{Detection Rate} = \frac{\text{# Defects found}}{\text{Total # Defects}}
\]

= Defects found in test process + Defects found afterwards (within certain time)

Not: All Defects in Application!!
Metrics for Planning

Number of test cases / fp: from measurements:

"Special" Cases:
- 1st time Testing, Exploring the Application
- Regression Testing, Application release N
Metrics for Control

Control: Adapt to Project Changes:
- Function Points: Skip/Add Appl.Parts
- Test Cases: Increase/Decrease Depth
- Defects: Postpone Repair, Reject Appl.Parts

(+ Regular Project Management Metrics)

Metrics for Determining the Stop Moment

Criteria:

1. Planned Test Cases executed?
   Sufficient for Estimated Detection Rate?

2. Expected New Defects Manageable?
   Using the Defect Curve
Metrics for Determining the Stop Moment

**Defect Curve**

Check Extrapolation with Real Data?
Use Results for Better Estimates?
Determine System/Project Profiles?
Measuring Software Reliability

VVSS2005
24 november 2005

Content

- Definition of reliability
- Principles of a reliability analysis
- Application
- Added value
Definition of “Reliability”

"The probability that an item will perform a required function without failure under stated conditions for a stated period of time”

Reliability growth curve

The graph shows the reliability growth over different intervals. The reliability increases with time, indicating improvements in the system's performance.
Statistical models

- Jalinski-moranda
- Musa basic
- Musa-Okumoto
- Littlewood-Verall
- Schneidewind
- Yamada

Selection of appropriate models

<table>
<thead>
<tr>
<th>assumption</th>
<th>atomic model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 time-between-failure</td>
<td>2</td>
</tr>
<tr>
<td>2 failure-count per testinterval</td>
<td>2</td>
</tr>
<tr>
<td>3 all testing intervals of the same length</td>
<td>3</td>
</tr>
<tr>
<td>4 software operated in similar manner as anticipated in operational use</td>
<td>4</td>
</tr>
<tr>
<td>5 all failures do not have same chance of detection</td>
<td>5</td>
</tr>
<tr>
<td>6 all failures do not have same chance of detection</td>
<td>6</td>
</tr>
<tr>
<td>7 all failures do not have same chance of detection</td>
<td>7</td>
</tr>
<tr>
<td>8 each failure is of the same severity as any other failure</td>
<td>8</td>
</tr>
<tr>
<td>9 failure detection rate forms a geometric progression and is constant between failure occurrences</td>
<td>9</td>
</tr>
<tr>
<td>10 failure detection is proportional to current fault content</td>
<td>10</td>
</tr>
<tr>
<td>11 failure rate remains constant over the interval between failure occurrences</td>
<td>11</td>
</tr>
<tr>
<td>12 the expected number of failures is a logarithmic (or proportional) function of time</td>
<td>12</td>
</tr>
<tr>
<td>13 the failure intensity decreases exponentially with the number of failures found</td>
<td>13</td>
</tr>
<tr>
<td>14 the cumulative number of failures detected at any time follows a Poisson distribution</td>
<td>14</td>
</tr>
<tr>
<td>15 the failure correction rate is proportional to the number of failures remaining in the program</td>
<td>15</td>
</tr>
<tr>
<td>16 the expected number of failures at the end of the interval is constant</td>
<td>16</td>
</tr>
<tr>
<td>17 the number of failures expected to be seen at the end of the interval is constant</td>
<td>17</td>
</tr>
<tr>
<td>18 the total number of failures expected to be seen at the end of the interval is constant</td>
<td>18</td>
</tr>
<tr>
<td>19 the total number of failures expected to be seen at the end of the interval is constant</td>
<td>19</td>
</tr>
<tr>
<td>20 the total number of failures expected to be seen at the end of the interval is constant</td>
<td>20</td>
</tr>
</tbody>
</table>

Total score: 36% 44% 25% 34% 46% 51% 60% 69% 100% 100%
Results (1)

Reliability for 8 hours

Results (2)

Reliability for 1 year for defects with severity =1

Reliability for 1 year = 0,9014
Results (3)

Cumulative number of defects

- Generalized Poisson
- Schneidewind: all
- Act. cumulative number of failures

Expected number of defects
In production = 5 of 6

Added value

- Increasing software reliability
- Support management in
  - Risk management
  - Service level management
  - Predict necessary capacity for incident management
- Cost saving in development and testing
Closed source application security testing

VVSS 2005

Dr. ir. Mario de Boer, CISSP
Collis BV
boer@collis.nl

Collis

- Founded 1997 in Leiden
- IT consultancy, project management, software development, quality assurance and testing
- Worldwide active in:
  - E-Business
  - QA & Testing
- Group Security Testing performs **structured** security testing:
  - Organizational audits
  - Network security testing
  - System security testing
  - Application security testing
Closed source

- In many cases we need to assess the security of applications without having access to the source code.

- Targets under test:
  - Suspected malicious software
  - (Popular) commercial software
  - Even interesting for software for which the source code is open

- Subjects
  - What do we want to determine?
  - Static versus dynamic testing
  - Automating the tests
  - Results

Target 1: Malicious software (malware)

Potentially any software of which you do not know:
- how it got installed on your system, and
- how it works

We wish to know:
- Known patterns
  - Is it known malware?
  - Does it contain known components?
- Entry points
- Payload
- Method for spreading
- Method of installation/infection
Target 2: Code security

Any code contains security vulnerabilities

We wish to know:

- Coding quality
  - Complexity of the code
  - Use of dangerous routines/coding constructions
- Entry points/attack surface
  - Parts of the application exposed to the outside world
  - Part of the application open to attack
  - User input, file access, API, network access, etc.
- Cryptographic strength

Methods of analysis (1/2)

- **Static analysis**: perform testing of the software without executing the software

- Examples:
  - Analyzing all words in a file
    - Advantage: trivial to execute
    - Disadvantage: limited results
  - Analyzing the calls to and from other code
    - Advantage: simple to execute
    - Disadvantage: limited results
  - Disassembling and code analysis
    - Advantage: very powerful results
    - Disadvantage: expertise is needed
Methods of analysis (2/2)

- **Dynamic analysis**: perform testing of the software by executing the software and monitoring the behavior

Examples:
- Monitoring of network connections, file and disk access, registry access, etc
  - Advantage: easy to execute
  - Disadvantage: limited in time and completeness
- Debugging: observing a running application
  - Advantage: very powerful results
  - Disadvantage: expertise is needed

Automatic static testing of software

1. **Software**
2. **Disassemble**
3. **Run testcases**
4. **Generate assembler code**
5. **Report**
6. **Analyse code**
7. **Dll, exe, etc**
Automatic static testing of malware

- Suspected software
- Known component comparison
  - Known malware is recognized
  - Preprocess
  - Remove security
  - Disassemble
  - Generate assembler code
- Analyse code
- Run testcases
- Report

Automatic dynamic testing of software

- Fuzz on all entry points
- Monitoring:
  - Access to files/disk
  - Access to registry
  - Access to network
  - Debugging
- Risks:
  - Software is being executed
Automatic dynamic testing of malware

Monitoring:
- Access to files/disk
- Access to registry
- Access to network
- Debugging

Risks:
- Malware is being executed

What can we achieve?
- A structured and repeatable way of analyzing the security of software
- Quick results
- A large part of the analysis can be performed automatically. For more detailed analysis manual analysis is necessary
Examples

- We will illustrate the techniques used in automatic static testing by giving some examples
  - Measuring complexity in an executable
  - Locating cryptographic routines
  - Locating exploitable weaknesses in executables

- The application of these techniques are dangerous, since these can just as well be used by malusers

Finally

- For questions and more information:

  Mario de Boer
  boer@collis.nl
Testing Security Issues Using Methods from Conformance Testing

Institute for Computer and Social Studies,
Dept. of Telmatics
Albert-Ludwig University,
Freiburg, Germany

Maike Gilliot

Overview: Testing

<table>
<thead>
<tr>
<th>Generation of test cases for functional conformance</th>
<th>Generation of test cases for security issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>• based on formal specification (SDL, ESTELLE)</td>
<td>• based on information about vulnerabilities (CERT, Bugtraq)</td>
</tr>
<tr>
<td>• based on specification in UML artefacts</td>
<td>• Techniques:</td>
</tr>
<tr>
<td>• Degree of automation:</td>
<td>– Fault injection testing</td>
</tr>
<tr>
<td>– addition of test objects</td>
<td>– Penetration testing</td>
</tr>
<tr>
<td>– addition of contracts</td>
<td>→ less formal</td>
</tr>
<tr>
<td>Advantages:</td>
<td>→ lower degree of automation</td>
</tr>
<tr>
<td>– evaluation of tests possible</td>
<td></td>
</tr>
<tr>
<td>– „higher“ quality of test cases</td>
<td></td>
</tr>
</tbody>
</table>
Importance of Security

Security is a self-contained attribute of software gaining importance:

- towards service-oriented systems
  - more connectivity, more attacks
  - integration of security is a precondition (cf. EVENT)

- evolving systems
  organizations (and thus their systems) are no longer stable, but adapting continuously to their environment (cf. Truex)

Goal:
„better“ security tests by generating them automatically

Security vs. functionality

Security:
Integrity, confidentiality, availability and accountability of data and communication

Security vulnerability: side effects of (unintended) functionalities that can lead to a violation of the security goals.

Ref: Whittaker: How to break Software Security
**Vulnerabilities: Classification**

- **Dependency:**
  An application relies on the network, the file system, other systems to work correctly.
  Is the application still secure if those components fail?

- **Context:**
  The same application can work differently depending on its context.
  Is the application in a different context still secure?

- **User interface:**
  Is the application secure when getting unexpected or malformed user input (sql-injection, buffer over-flows)?

- **Violated execution order:**
  Is it possible to execute actions that are not secure in this state?

---

**Adding Information: Misuse Cases**

**Idea:**
Add security relevant information to the specification using misuse cases:
1. Add misuse cases to specification,
2. derive test cases and
3. invert the test verdict

**Misuse Cases:**
- Used to elicit non-functional requirements
- Negative scenarios to look at the application from an attackers' point of view.
By example

- Given: specification

- Add some misuse cases for Vulnerabilities
  - due to dependencies
  - due to context
  - due to malformed or unexpected inputs
  - due to violations of the execution order

Conclusion

- Degree of automation is limited:
  - misuse cases have to be added manually:
  - to find misuse cases means to find security vulnerabilities. This can (probably) not be done automatically (cf. Firesmith, FMEA)

- Difficult to add misuse cases for all types of vulnerabilities:
  - a specification is suppose to represent an application (and not its environment, ist context or its dependencies)

  Specification may not be the right basis for security tests
Next steps

Use vulnerabilities as basis for security tests

1. Model vulnerabilities
   • which information necessary?
   • choose/extend/build model

2. derive security test out of this vulnerability model

The End

Thank you!

Questions?
Literature

- Bugtraq: Vulnerability database, source: http://www.cert.org/nav/index_red.html
- CERT Vulnerability database, source: http://www.cert.org/nav/index_red.html
Introduction: Quality of Software

- Quality of software can be split into different aspects and techniques:
  - testing, debugging, verification, validation, metrics, refactoring, reviews, coding guidelines …
  - only a few techniques are employed outside university, e.g. reviews and simple unit- or integration tests

- Problems
  - lack of (experienced) employees familiar with concepts of software quality
  - strict time and monetary constraints in development department
  - how to measure the quality of the quality assurance? („Manual testing of software is self-contradictory.” Beizer 1990)

- our primary focus: automated testing of software
Introduction: Testing

- Software testing:
  - Test criteria
  - Generation of test cases
  - Saving of test cases
  - Execution of test cases

- Test criteria
  - Functional (black-box)
  - Structural (glass-box)

- Generation of test cases
  - Random
  - Dynamic
  - Static

Generating Test Cases Using a Symbolic Virtual Machine

Symbolic Execution: Running Example

- Symbolic execution
- Static analysis
- Features of (functional) logic languages
- Virtual machine
- Constraint solving

Running example: Binary search

```c
static int binsearch(int[] a, int low, int high, int x)
{
    int mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (a[mid] < x) low = mid + 1;
        else if (a[mid] > x) high = mid - 1;
        else return mid;
    }
    return -1;
}
```
Symbolic Execution: Design Decisions

- byte code vs. source code
  - influence of compiler
  - several languages compiling for the same virtual machine
  - stability of specification

- language of our choice: Java
  - object oriented
  - stability of the specification of the VM
  - is used in theory and practice in our department

- Java virtual machine
  - stack-based
  - intermediate code
  - platform independent
  - simple language

→ "binsearch" in byte code

Symbolic Execution: Byte Code

```
0: iload_1  
1: iload_2  
2: if_icmpgt 47
5: iload_1  
6: iload_2  
7: iadd   
8: iconstant_2  
9: idiv   
10: istore 4  
12:aload_0  
13:iload 4  
15: iload   
16: iload_3  
17: if_icmpge 28
19: iload 4  
20: iload_1  
22: icstant_1  
23: iadd
```

Generating Test Cases Using a Symbolic Virtual Machine

Roger A. Müller, Christoph Lembeck, Herbert Kuchen
Symbolic Execution: Byte Code (cont.)

```java
20: iload 4
22: iconst_1
23: iadd
24: 1store_1
25: goto 0

0: iload_1
1: iload_2
2: if_icmpge 47

47: iconst_m1
48: ireturn

36: iload 4
38: iconst_1
39: iadd
40: 1store_2
41: goto 0

28: aaload_0
29: iload 4
31: 1aload
32: iload_3
33: if_icmpge 44
```

Symbolic Execution: Basics

expressing variables w.r.t. constants and input variables:

```
mid=(low+high)/2
```

Generating Test Cases Using a Symbolic Virtual Machine

Roger A. Müller, Christoph Lembeck, Herbert Kuchen
Symbolic Execution: Branching

- **Branching depending on**
  - alternatives
  - test criterion
  - evaluation of constraints

- **Branching can occur on**
  - im- or explicit exceptions
  - conditional jumps or switches
  - method invocations

- **Each branching adds a constraint to a global constraint system describing the current path w.r.t. to input parameters and constants**

- **Needed for branching**
  - constraint solver system
  - test criterion
  - backtracking to cover all applicable paths

---

Symbolic Execution: Branching (cont.)

- Each branching adds a constraint to a global constraint system describing the current path w.r.t. to input parameters and constants

- **Examples:**
  1. **if (a>3) then B else C**
     - Execution of B: a>3 on stack
     - Execution of C: a<=3 on stack
  2. **if (a>3) then B else if (a>1) then D else E**
     - Execution of: a>3 on stack
     - Execution of D: a<=3^a>1 (i.e. 1<a<=3) on stack
     - Execution of E: a<=3^a<1 (i.e. a<1) on stack

- **Preconditions:**
  - constraint solver system
  - test criteria
  - backtracking capabilities
  - branching strategy
Symbolic Execution: Branching (example)

Symbolic Execution: Constraint Solver System

- **duties**
  - compute result
    - input variable values
    - return value if applicable
  - feasibility check
    - system still solvable when adding a constraint

- **capabilities**
  - linear / non-linear constraints
  - integer / non-integer constraint
  - symbolic / numeric computation
  - equalities / inequalities
  - constraint solver manager
    - iterative solution
    - classification and breakup of constraint
    - choosing an appropriate constraint solver
Symbolic Execution: Constraint Solver System (cont.)

- SJVM
- Normalizer
- Constraint Stack
  - Stack Element
    - Constraint System
      - \( y \geq 8 \)
      - \( x \leq 10 \)
    - Constraint System
      - \( y = 3 \)
      - \( x = 0 \)
      - \( x = 10 \)
- Simplifier
- SolverManager
- Controller
- Constraint Solver Selection Unit
- BooleanSolver
- GaussianSolver
- EliminationSolver
- SimplexSolver
- BuchbergerSolver
- BisectionSolver
- NewtonSolver

Symbolic Execution: Backtracking

- current execution ends:
  - result
  - invalid path
  - uncaught exception

- backtracking
  - well known from the implementation of functional-logical programming languages
  - return to prior program state, e.g. to a branching instruction, method invocation
  - naïve approach: just copy the whole program state, better approach:
    - choice points: save information about the prior program state (e.g. program counter, pointer to constraint system, trail ...)
    - trail: save prior state of a variable (stack element ...) once at the first change of value
Symbolic Execution: Backtracking (example)

Generating Test Cases Using a Symbolic Virtual Machine

Roger A. Müller, Christoph Lembeck, Herbert Kuchen
Symbolic Execution: Arrays

0: iload_1
1: iload_2
2: if_icmpgt 47

binsearch(int[] a, int low, int high, int x)

Initialisierung: int[] = null

5: iload_1
6: iload_2
7: iadd
8: iconst_2
9: idiv
10: istore_4
12: aload_0
13: iload 4
15: iload
16: iload_3
17: if_icmpge 28

NullPointerException

Symbolic Execution: Arrays

0: iload_1
1: iload_2
2: if_icmpgt 47

binsearch(int[] a, int low, int high, int x)

Initialisierung: int[] = new int[0]

5: iload_1
6: iload_2
7: iadd
8: iconst_2
9: idiv
10: istore_4
12: aload_0
13: iload 4
15: iload
16: iload_3
17: if_icmpge 28

ArrayIndexOutOfBoundsException
Symbolic Execution: Arrays (cont.)

0: iload_1
1: iload_2
2: if_icmpgt 47

binsearch(int[] a, int low, int high, int x)
Initialisierung: int[] = new int[1]

5: iload_1
6: iload_2
7: iadd
8: iconst_2
9: idiv
10: istore_4
12: aload_0
13: iload_4
15: iaload
16: iload_3
17: if_icmpge 28

Annahme: mid = 0, dann in 15 keine Exception

Summary

- open ends
  - binary shifts
  - concurrent programming
  - precision of calculation
  - state explosion
- summary
  - tool generation of structural tests
  - using elements of functional-logical programming and a constraint solver (manager)
  - flexible test criterion
  - prototype: Eclipse plug-in
ASM-based Run-time Verification of Application Protocols

Jan Smans, Bart De Win, Wouter Joosen, Frank Piessens
K.U.Leuven

André Mariën, Johan Van Oeyen
Ubizen

Overview

• Problem Statement
• Background
• Run-time Verification of Application Protocols
  – Mapping application events to actions
  – Implementation and Tool support
• Deriving model programs from valid client code
• Conclusions and Future Work
Problem Statement

- Correct (secure) functioning of a distributed application can depend on the client adhering to an (often implicit) protocol or workflow
  - Stateless session beans on the application server
  - Protocol coded in web tier or client tier

Problem Statement

- Example: prototypical e-commerce application
  - Stateless service methods
    - Product lookup: `Product lookup(Product key)`
    - Process payment: `void processPayment(String customer, int amount)`
    - Ship products: `void shipProducts(String customer, Basket b)`
    - Compute price: `int computePrice(Basket b)`
  - Expected client operation
    1) lookup several products and possibly put in basket
    2) compute price of basket
    3) ask confirmation from the user
    4) process payment for computed amount
    5) Ship all products in basket
Problem Statement

• If client workflow is enforced through coding, there can be a substantial risk that workflow logic is changed or bypassed
  – Example: workflow is implemented in a web tier
    • Can be bypassed through forceful browsing or parameter tampering
  – Example: workflow is implemented in client tier
    • Can be bypassed through reverse engineering of, and tampering with client-side components

Problem Statement

• How can we ensure that a (possibly remote) component of a distributed application adheres to an (implicitly or explicitly) expected protocol?
  – With a focus on application-level protocols in object oriented languages (C#, Java, ...)

• Our approach:
  – Position a (configurable) filter right in front of the application server
  – What is a good language for specifying the behavior of the filter?
Background

- **Model automata [1]:**
  - Are a variant of Abstract State Machines
    - states are first-order structures
  - Can be thought of as Labeled Transition Systems
    - actions look like (atomic) method invocations
- **Model programs**
  - Compactly encode large automata
  - Programmed in Spec# [3] or AsmL
- **The Spec Explorer tool [1,2]:**
  - Compiles model programs to .NET assemblies
  - Provides support for exploring the state space
  - Supports model-based test case generation

Overview

- Problem Statement
- Background
- Run-time Verification of Application Protocols
  - Mapping application events to actions
  - Implementation and Tool support
- Deriving model programs from valid client code
- Conclusions and Future Work
Run-time verification

General approach:
- Position a filter between the two components at the side of the trusted component
- Filter is programmed with a model program
- If an observed application event does not correspond to a possible action in the model program, defensive measures are taken
  - Abort session
  - Log event

Mapping application events to model program actions

- Choice 1: action = method invocation
  - Relatively easy to write
  - In some cases inappropriate
    - Non-atomic method bodies (e.g. callbacks)
    - Full method invocation information is only known upon return of a method, hence too late to block method call
- Choice 2: action = method entry or method return
  - Most general
  - Harder to write good model programs
- In this presentation, we focus on choice 1
Example Model Program

```java
enum ShoppingState {ProductSelection, ReadyToPay, ReadyToShip, End};
ShoppingState state = ShoppingState.ProductSelection;
int topay = 0;
Product product = null;

public Product LookupProduct(String key) requires state == ShoppingState.ProductSelection; {
    return <call real product lookup here>; }

public int ComputePrice(Product p)  requires state == ShoppingState.ProductSelection;   {
    state = ShoppingState.ReadyToPay;
    topay = <call real price computation logic here>
    product = p;
    return topay;   }

void ProcessPayment(int amount)  requires state == ShoppingState.ReadyToPay;
requires amount == topay;    {
    state = ShoppingState.ReadyToShip;
    <call real process payment> }
```

(Part of) Corresponding Model Automaton

(graph generated with the Spec Explorer tool from Microsoft Research)
Implementation & Tool Support

• Spec Explorer
  – Explore specified application protocol
    • Is this desired client behavior?
  – Automatically instrument code to call the model program
  – Compile the model program to a .NET assembly that implements the desired filter

• Improving performance
  – If the state space is finite, a much more efficient implementation can be generated that checks the FSM generated by Spec Explorer

Overview

• Problem Statement
• Background
• Run-time Verification of Application Protocols
  – Mapping application events to actions
  – Tool support

• Deriving model programs from valid client code
• Conclusions and Future Work
Deriving model programs from client code

- Writing the model programs can be labor-intensive
- An implementation of a valid client is also a specification of the protocol
  - But possibly an over-specification
  - Not immediately usable to program a filter
- Our approach
  - Compile a non-deterministic pseudo-code client program to a model program
  - Possibly compact the generated model automaton through state grouping

Example pseudo client code

```java
Product p;
bool buy;
String key;
int price;

key = choose String;
p = lookupProduct(key);
price = computePrice(p);
buy = choose bool;
if (buy) {
    processPayment("XYZ", price);
    shipProducts("XYZ", p);
}
```

Client Program State: values for vars + PC

All action sequences generated by this non-deterministic client program are allowed
Construction of corresponding model program

- Client program state (cps):
  - Values for variables + Program Counter (PC)
  - Is stable if the PC points to a method call to the server

- Client State CS in model program
  - Set of possible stable client program states

- Precondition for a method call
  - There exists a cps in CS that is ready to perform that call

- Client State update after a call
  - Filter all cps that don’t have the observed call enabled
  - Reduce all remaining cps’s to a stable cps

Discussion

- Advantages:
  - Pseudo client programs are much easier to write,
  - … and can even be derived from existing client code by replacing user input with choose statements
  - Pseudo client programs have a “default deny” semantics

- Disadvantages:
  - Pseudo client programs sometimes over constrain the protocol
  - Performance of the filter is much worse
    • But this could in principle be improved through more advanced compilation techniques
Conclusion and Future Work

• Model-programs are a suitable specification formalism for application protocols
  – Easy to implement protocol checking filter
• Model-programs can be written as independent specifications of the application level protocol
• Or model programs can be derived from existing client code
• This is work in progress:
  – Implementations are only in the prototype stage
  – Correctness proofs for the derivation of model programs have yet to be constructed

References

Introduction Deloitte Enterprise Risk Services

### Assurance
- Enterprise Risk Management
- Process and CRM/MCS/ERP assessment and Project QA
- Data Quality Improvement
- Information Security Assessments
- Operational & IT audit support & cosourcing

### Control & security solutions
- Process & System Management incl. Integrity Control & Security
- Information Security Solutions
Introduction

- Business
  - is for entrepreneurs

- Business Management
  - is for MBA’s to make entrepreneurs more effective

- Business Process Management
  - is a tool for managers to improve business operations effectiveness

- Business Process Reliability Management (or Business Process Control)
  - Guarantees reliable structured operational management information (information quality)
  - Turns Internal Control Compliance into Competitive Advantage

Why business process management

- NEXT WAVES IN BUSINESS MODEL DEVELOPMENT
  - ERP SYSTEMS BECOME “SERVER FARMS” AND ARE NOW EXTENDED WITH
    - STANDARD MISSION CRITICAL SOLUTIONS (mcs)
    - CUSTOMER RELATION MANAGEMENT (CRM)
  - SUPPLIER AND CUSTOMERS INTEGRATED IN THE PROCESS CHAINS
  - BUSINESS PROCESS OUTSOURCING (Focus on Core Business)
  - CENTRALISATION OF SUPPORTING FUNCTIONS IN SHARED SERVICE CENTERS
  - USE OF MARKETPLACES AND PORTALS

- MANAGEMENT NEEDS OF THE PROCESS ENTERPRISE
  - ASSURING STAKEHOLDER VALUE
  - FINETUNING THE ORGANISATIONAL MODEL
  - SUPPORTING THE PROCESS ENTERPRISE
  - TECHNICAL APPLICATION MANAGEMENT & SUPPORT + INFRASTRUCTURE OPERATIONS
**Why business process management**

**I. Operating Model**
More efficient processes and operations to position the Company for the future

**II. Organisation**
Well positioned & focused organisation to support future growth and profitability

**III. Process Management Plan**
Tactical plan for mobilising and managing process performance realising enduring improvements beyond the quick hits

**IV. Performance Management & Metrics**
Tools & Dashboards that will assist and focus management in achieving on-going performance improvements

---

**A Critical Driver of Investor Trust: Financial Information Quality**

“Trust” is a key driver of earnings quality

- Transparency
- Timeliness
- Accuracy
- Predictability

Companies may improve performance addressing the components Information Quality

- Earnings Quality now competes with earnings growth

---

**Financial Information Quality**

<table>
<thead>
<tr>
<th>Performance Measure</th>
<th>Month 1</th>
<th>Month 2</th>
<th>Month 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Financial Ratio 1</td>
<td>10%</td>
<td>10%</td>
<td>8%</td>
</tr>
<tr>
<td>Financial Ratio 2</td>
<td>8%</td>
<td>2%</td>
<td>3%</td>
</tr>
<tr>
<td>Operational Ratio</td>
<td>10%</td>
<td>5%</td>
<td>3%</td>
</tr>
<tr>
<td>Total</td>
<td>10%</td>
<td>5%</td>
<td>8%</td>
</tr>
</tbody>
</table>
After the implementation most organisations do not pay specific attention to process monitoring & management…

**Deployment phase**

1. Process Scoping
2. Process Modelling
3. Process Implementation
4. Process (Control) Monitoring
5. Competitive advantage

**Re-implement**

Most organisations stop here!!

**Implementation phase**

Opportunities for Process improvement / process innovation and design of business controls

Align focus and scope of the organisation and its business processes

…while business process reliability management and controls compliance monitoring can be implemented in different ways.

- **Qualitative self assessment**
  - Employees answer questions on their own work
  - Simultaneous they are updated on best practices
  - And identify bottlenecks and solutions

- **Quantitative assessment**:
  - Focus on “hard” metrics (what gets measured gets done)
  - Less organisational change component due to less direct involvement of individual employees
Business Process Control (or business process reliability management) results in 

- Uniformity and transparency
  - Uniform business procedures
  - Uniform results
  - Employees share information
- Better control on operations and management information
  - Uniform management information
  - Specific information regarding bottlenecks and issues
  - Business Processes on the management agenda
  - Comply with regulations (incl. SoX)

- Motivation of employees
  - Commitment of employees
  - Employees start to deploy the business processes
- Continuous business process improvement (More effective and efficient processes)
  - Relevant information any time any format on subjects that are regarded as important
  - Think and communicate business processes is the basis for improvements

.. turning business process management and process monitoring into competitive advantage

My Business Process Enablement Layermodel

Implementation phase

Manual business process procedures

Automated business process procedures (Workflow)

Implementation of application functionality

Master Data/Coding implementation

ERP/eBusiness “LEGO” standard components

ADD ON

Build in controls and security

Deloitte
My Business Process Enablement Layermodel

Deployment phase

Business Process (Controls) Compliance Monitoring

- Manual business process procedures
- Automated business process procedures (Workflow)

Implementation of application functionality

- Master Data/Coding implementation
- ERP/ eBusiness “LEGO” standard components

ADD ON

Build in controls and security

Layermodel for IT-architecture

- Employee portal
- Supplier & Customer portals
- Management stakeholder portal
- Process portal
- Mail

- CRM, Mission Critical Systems (MCS) ERP
- Process - Modelling, - Performance Management and -Compliance Monitoring

- Business Information Warehouse
- Knowledge Management Warehouse
- Process Performance Warehouse

BEP
Business Exchange Platform

BOP
Business Operating Platform

BIP
Business Information Platform
Layermodel for Process-architecture
Example processes in a Metro Organisation in a big city

Business Process Control Framework

Business Process Control Framework
Internal Control Framework
SOX/ 8th Directive requirements
Risk Driven

Areas of Focus
Business Process Control
ERP Security & Authorizations
Data Integrity
Information Technology
Integrity

Direction of Control
Detective
Preventive

Components of Control
Control Awareness
User Procedures
Policies and Standards
Training

Effectiveness, Efficiency, Reliability, of Information

(Internal) Audit Approach (IA)
Audit Methodology
State of the art tools

Business Drivers
Risk Management
KPI’s/PPI’s/CPI’s
Business Balanced Scorecard

Aspects of Quality
Validity
Audibility
Accuracy
Efficiency
Effectiveness, etc

Business Process Control Methodology

Deloitte
COSO Internal Control Framework

Monitoring: The process to determine whether internal control is adequately designed, executed effective and adaptive

Control Activities: The policies and procedures that help ensure that actions are identified to manage risk and are executed and timely

Risk Assessment: The evaluation of internal and external factors that impact an organization’s performance

Information and Communication: The process which ensures that relevant information is identified and communicated in a timely manner.

Internal Control Cost Drivers – Management’s Tasks

- For the auditor to satisfactorily complete an audit of ICFR, management must do the following:
  - Accept responsibility for the effectiveness of the company’s internal control
  - Evaluate the effectiveness of its internal control over financial reporting using suitable control criteria
  - Support its evaluation with sufficient evidence
  - Present a written assertion about the effectiveness of its internal control over financial reporting

- Disclaimer in case of non compliance
Lessons from Business Process Control Projects

- Understand the real size of the project cultural differences
- Pilot project or experience an absolute must
- Documentation always inadequate document management
- Tool required to collate and report on control status
- Strong arguments to centralise and standardise
- Costly compliance or efficient governance?

Deloitte
What is the competitive advantage

- **Standardized and documented processes enable**
  - Uniformity and transparancy (across departments and organisations)

- **Build in controls and security enable**
  - Better control on operations and management information (more reliability)

- **Process compliance monitoring enables**
  - Motivation of employees
  - Continuous business process improvement (more effective and efficient processes)
Conclusion: Get Your Business Processes Measured, Monitored, Managed, Mature

in order to Turning Internal Control Compliance into Competitive Advantage

• Processes and Data

» survive applications and organisational models
» should be under User control (CPO organisation)
» should be supported by an integrated process management portal
» control, risk management, quality and (internal) audit functions should leverage the semantics and documentation
» process modelling is one, managing process performance and continuous proces improvement is 2,3,4 …n

To be a Business Process (Control) Pilot requires a long horizon, a reliable engine, adequate training, a professional crew, and a strong hand on the controls …

…………in order to assure a smooth flight and a safe landing and arrival
Gestructureerd accepteren van bedrijfsprocessen

Klaas Smit

Atos Origin, Groningen, the Netherlands
LOOKING FOR STABILITY

Cornelis Huizing, Ruurd Kuiper, Teade Punter, Alexander Serebrenik
Laboratory for Quality Software (LaQuSo)
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{C.Huizing, R.Kuiper, T.Punter, A.Serebrenik}@tue.nl

ABSTRACT: Tools ranging from code structure metrics to assertion checking are applied to assessment of stability under future modification. The example code and documentation concern a moderately-sized but realistic Java implementation of a safety layer for a model train system. The experiences show added value of combining tools and, somewhat surprisingly, indicate that assertion checking tools not only provide positive information but also help in finding errors that would go unnoticed even applying exhaustive approaches like model checking.

Keywords: stability, ISO 9126, software product assessment, static analysis, tools.

1 INTRODUCTION

We present experiences with computer-assisted assessment of the stability under modification of software, by means of static analysis. The experiments are carried out in the context of the laboratory of Quality Software (LaQuSo) at Eindhoven University of Technology (TU/e), The Netherlands. One of the aims of LaQuSo is to assess code from industry, e.g., for certification.

Static analysis on the code is appropriate, because for stability it is code properties that are decisive rather than the behavior per se. Furthermore, not only code but also documentation is relevant for stability. Tool support serves two different purposes: First, it provides the efficiency to make it feasible to assess larger code, second it provides the rigor that is necessary for validation and certification purposes. We therefore think it justified to try to draw conclusions that, although being directly based on our experiments, extend to product software.

Our contribution is twofold. First, we present an operationalization of stability assessment by means of five static analyses. Each of the analyses has been carried out using an out-of-the-box tool: Sotograph, SA4J, IntelliJ IDEA, Gemini, and ESC Java. The tools range from high level assessment of the structure of code to lower level checking of code against assertions from the specification. Second, we assess the feasibility of the approach by performing a case study.

We present a case study concerning the assessment of the stability of a model train security system, written in Java. The software and its documentation were developed as a student software engineering project at TU/e.

On the whole we are cautiously positive about the possibilities provided by the tools we used. We mention two general observations. First, we experienced that it is advantageous to use the tools in an incremental fashion. Order of application is important: e.g., some structural properties are prerequisites to make assertion checking feasible—it makes for efficient use of tools to find out what is feasible quickly, and early on. Furthermore, the combination of results from different tools shows that Aristotle’s adage “the whole is more than the sum of its parts” applies. For example, the development process can be assessed, without this assessment being a specific result of one tool.

Second, we found that especially the assertion checking tool is not only, as one might suppose, useful to show rigorously that code does satisfy specified properties, but also reveals faults: because of the fine grained modularity of the checks, the faulty code can be closely identi-
fied. In particular, some malpractices detrimental to stability where brought to light that no behavioral check, even an exhaustive approach like model checking, would have exposed.

The paper is structured as follows. Section 2 introduces the notion of stability, based on the ISO standard. Five stability related issues are identified. In Section 3 the case is described. The most extensive Section 4 contains the experiences. This section is organized according to the stability-related issues identified in Section 2. In Section 5 we conclude.

2 STABILITY

Assessing software quality is a difficult task. To formalize the intuition of software being good or bad the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have proposed a series of standards [4, 6, 7, 5]. The standards specify product quality characteristics, such as functionality, reliability, efficiency and maintainability, divide them further into subcharacteristics and suggest methods of evaluating them. For example, maintainability is, according to [4], the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, requirements, and functional specifications. Maintainability is further divided into analyzability, changeability, stability, testability and maintainability compliance. In this paper we focus on software stability.

Definition 1 ([4]) Stability is the capability of the software product to avoid unexpected effects from modifications of the software.

We consider quality of the software itself, so called internal quality [7], rather than quality of the computer-based system including the software [6] or effects of using the software in a specific context of use [5].

ISO 9126-3:2003 specifies two stability metrics measuring internal quality, namely change impact and modification impact localization. Both metrics are based on calculating some value after a modification. Unfortunately, this information is not always readily available. For instance, not all pre-release modifications of the software might be kept. Moreover, documentation may be missing, incomplete or inconsistent with the implementation. Therefore, the ISO-recommended metrics are not applicable and we need a way to infer stability-related information from the software implementation.

Instead, we identify the following five stability-related issues. These issues are intended as a reasoned attempt to operationalize the ISO definition of stability. Two ideas guide our reasoning: We consider out-of-the-box tools that provide information that appears relevant to stability. We aim to cover the most important levels at which design and implementation can be assessed. More experiments and evaluation are needed to quantify how well they correlate with the definition.

The stability-related issues are then the following. Each of these issues is addressed for the case study in a separate subsection of Section 4.

- functional decomposition. By functional decomposition we understand division of the system in a number of independent but cooperating units. In particular, we are interested in detecting discrepancies between the functional decomposition as presented in the documentation and as implemented in the software.

- coupling. By coupling we understand a degree of interdependence between a pair of units. By coupling we understand a degree of interdependence relations between a pair of units. For the analysis of such interdependencies we have a look on interface descriptions and compare those descriptions as they were documented with how they are actually implemented. We focus on call relations (inbound and outbound calls) on package level, because this will provide us information about the intensity of the relationship on an adequate abstraction level.

- dependency structure. By dependency structure we understand the entire system of relationships between different units of the system. For instance, it is well-known that a modification in a tangled unit, i.e., a unit belonging to a set of units such that any of them depends on any other one, is likely to diffuse through the entire tangle.
code duplication. By code duplication we understand presence of identical or almost identical code fragments. By “almost identical” we understand minor syntactical differences between the fragments such as renaming variables. Introducing modifications into one instance of the duplicated code necessitates propagation of the modification to other instances.

- implementation malpractices. Some malpractices do not affect code functionality and reliability but code stability. In presence of such malpractices, for instance, in object-oriented languages adding a new class that inherits from the existing one can lead to unexpected behavior of the resulting system.

The first three issues can be viewed as features of the design, the latter two more directly belong to the realm of implementation.

As means to assess each of the characteristics above we opt for static analysis techniques, i.e. analysis on the code rather than the more usual dynamic analysis on the behavior of the running program. The motivation is, that for stability it is code properties that are decisive rather than the behavior per se.

3 CASE STUDY

The software we have chosen as a case study implements a safety layer for a Märklin model railway system. The railway system consists of a number rail tracks, which can include switches and turnouts. The railway topology has been fixed; a rough idea of its complexity can be obtained from Figure 1. At every moment of time up to eighty model trains can ride simultaneously on the railway. The user can manually operate the system by providing commands like “add a new train”, “turn the lights off” or “prohibit an entry to a rail track”. The safety layer takes care of minimizing the number of collisions and derailments. Moreover, it enforces the riding trains to move with the maximal speed that does not contradict the safety requirements.

The software has been developed by a team of eight third-year students as a part of their software engineering assignment. The implementation has been done in Java and Delphi. We have restricted our attention to the Java part, which consists of 9 packages, 164 classes and counts 17828 lines of code. The implementation makes use of seven different API packages, including java.nio.channels.* and javax.comm.*. As part of the assignment, the students also provided a Software Requirements Document (SRD) and an Architectural Design Document (ADD).

Stability assessment has been required due to the intention to reuse the implementation as the basis for a more advanced train management system assignment.

4 ANALYSIS

4.1 Functional decomposition

A proper functional decomposition is an important factor in the quality of software. It enables to handle complexity by distributing functionality over several components. We did not find tooling for a direct objective quantitative measure for the structural quality functional decomposition of software. However, in the design phase a proper functional decomposition should be defined. This can be compared to the decomposition in the actual implementation. Discrepancies between these two are a potential cause for instabilities. One reason for this is that undocumented deviations from the design are often violations of architectural rules and causes of increased complexity. Another reason is that later changes based on the documentation can have unexpected effects if the documentation is inaccurate.

In the case study software, the documented functional decomposition into packages could not be compared with the package structure of the actual software. The reason is, that the relationships between packages in the former are use relations and in the latter part-of relations. Therefore, we used the tool Sotograph [11] to derive the package communication diagram. Comparing the diagrams reveals some differences: one package has fewer connections than documented, some packages have more connections.

The overall picture is that the package decomposition corresponds to the documented functional decomposition. It is difficult to make strong claims about the functional decomposition. When more information about the
Figure 1: Railway topology
intended architecture is available, a better assessment can be made about the design itself and also as to how well this is implemented.

### 4.2 Coupling (Interfaces)

Given a structural decomposition, further assessment can be done on the coupling between programming elements as well as on the (internal) cohesion of such elements. Coupling is the degree of interdependence between programming elements, i.e. modules, packages or classes. It is an attribute of a set of pairs of these elements, rather than of a complete design. Low coupling between elements is desirable in general, as it allows a divide and conquer approach to complexity; it enhances stability in particular, because modifications on one element have little effect elsewhere. Cohesion concerns the interdependence inside a programming element. High cohesion is desirable, as it confirms that only strongly dependent material is grouped together.

We assess coupling only; a similar assessment of cohesion would be possible, but we expected most insights in the tool practicability to show up already for coupling.

In the case study, coupling is expressed as between particular packages X and Y. The interdependencies can be of several types, such as calls, polymorphic calls, reads and type accesses. For our analysis several coupling relationships were examined by comparing the interface design documentation as planned to be implemented (see Figure 2) and the call graphs of the implemented code.

The call graph on package level is produced with Sotograph (see Figure 3). The call graph on class level can be viewed as a refinement of the call graph on package level: the nodes are classes and packages, respectively.

Starting with the interface design document, we expected to find coupling relationships between the packages Train Control, BSIinterface, Security and HAL (hardware abstraction layer). These packages call each other and provide data according to a layered pattern. BSIinterface and HAL are the interfacing packages with the rest of the packages. Commands are sent from Train Control to Security, after which a confirmation or eventually an error message is sent from Security to Train Control. Then a next Command can be sent. In addition, events will be sent that report about what is happening in the traffic and the Security layer. There is also communication between Security and Configuration, which concerns a logging facility for maintenance purposes. A third type of communication is for error handling and concerns Exceptions and the rest of the packages.

Figure 3 provides the Call graph of the implemented code as produced with Sotograph. Looking at the coupling types (the arrows) we found call relationships between the packages Train Control, BSIinterface, Security and HAL. These call relations are set according to what we expected to find in a layered pattern of communication. Also the communication between Exceptions and the rest of the packages is according to what we expected to find. There are many throws of exceptions between the methods of the concerned classes and packages. Exceptions are thrown if a precondition of a method is not valid. The methods that receive such thrown exceptions should send it further (throw) to the calling method to deal with it or handle it themselves (catch). However, looking at the third type of communication, between the packages Configuration and Security we see that the first package is not exclusively coupled to the Security package as it was planned. In fact the package is related to many more packages: Train Control and HAL are also communicating with the Configuration directly, instead of via the Security package. The intertwined coupling relationships that result from this makes the communication harder to understand and we regard this as a negative impact on the stability of the system. On behalf of these findings we conclude that the overall coupling relationships in the system concern exchange of data. They are well designed and the implementation is rather good. A clear interface between Configuration and other packages is missing. From a coupling perspective of the system, the systems Stability is well enough, not good.

### 4.3 Dependency structure

In this section we consider a number of malpractices related to software architecture as regards the dependency structure. Presence of these malpractices can bear witness of a problematic design or of a violation of the original design.

By a structural malpractice, also called an anti-pattern, we understand a system of inter-element dependencies that facilitates propagation of a change. A typical
Figure 2: Calls according to the documentation

Figure 3: Calls according to the implementation
example of such a malpractice would be a so-called local breakable, an element such that many other elements depend on it. In such a case, when the element is changed, elements that depend on it might require modifications. Local breakables are typically undesirable because they “know too much”. In order to improve stability it is advisable to refactor a local breakable into several elements to distribute its dependencies. A dual notion is a notion of a local butterfly, i.e., an element that immediately depends on many other elements. Typical examples of local butterflies in Java are basic interfaces, abstract base classes, or utilities. Local butterflies are not necessarily problematic, but in an unstable system changes can affect areas beyond immediate notice. Local breakable which is also a local butterfly is called a local hub. Local hubs are typically undesirable because they amplify the effects of change throughout the system. The global counterparts of the notions of a breakable, a butterfly and a hub consider transitive closure of the “depends” relation. Similarly to local butterflies, global butterflies in Java are usually interfaces or utilities. Global breakables typically are implementations of the highest-level concepts in a system. Except for high-level concrete implementations, global breakables are generally undesirable because they indicate lack of modularity in the system. Global hubs are very harmful and indicate a poorly conceptualized, unstable system. Global hubs imply that the entire system is entangled and interdependent. Small changes can have ramifications that spread throughout the system. Finally, a tangle is set of elements such that a change in one element can affect all other elements. Tangles are known to be a major cause of instability in large systems. Therefore, there should be no tangles of more than two elements. Based on the discussion above we classify local and global butterflies as less important anti-patterns, local breakables, global breakables and local hubs as important anti-patterns, and global hubs and tangles of more than two elements as very important anti-patterns.

Architectural malpractices introduced above can be viewed as parameterized by the interpretations of “an element”, of the “depends” relationship and of the “many” threshold. Provided that we work with an object-oriented language we consider packages and classes as elements. One can consider many different kinds of depends relations, such as accesses, calls, contains, extends, implements, instantiates, references, throws or uses. For example, when a class A contains an instanceof-test or a casting to a class B we say that A references B. Threshold values for different kinds of malpractices might depend on configuration of a measurement tool.

To discover presence of anti-patterns in the case study software we have used a freely-available tool called SA4J, abbreviating “Structural Analysis for Java”. The tool has been developed at IBM and can be downloaded from [10]. Table 1 summarizes threshold values of SA4J for different kinds of anti-patterns and results of the application of the tool to the case study software. For local anti-patterns the threshold values are absolute, while for the global ones they are expressed as percents of the total number of elements. It should be noted that for hubs two threshold values should be taken into account: the in-threshold and the out-threshold.

SA4J discovered that one of the tangles involves 24 classes and packages (17%). The “depends” relation discussed above allows to measure stability as a function of an average number of elements affected by a modification of one element, where affected should be understood as a transitive closure of “depends”. Formally, stability metrics calculated by SA4J is the percentage of elements that are not expected to be affected by a change. For highly-stable systems this value should exceed 90%. For the case study software the value of the metrics was 65%, far below the desired threshold.

Summarizing the discussion above, we observe that the system contains a significant number of important anti-patterns (tangles, global hubs, global and local breakables) and that the dependency metrics is out of the stability boundaries. Therefore, we can conclude that from the architectural perspective stability of the case study software is poor.

### 4.4 Code duplication

Code duplication is a known problem in software development. Generally, speaking when one of the instances changes, the modification has to be propagated to all other instances although the instances do not “depend” on each other. Moreover, some of the architectural anti-patterns can be eliminated by code duplication without actually improving the design. For instance, if A is a local breakable that depends on classes B, . . . , Z one
might have replicated $A$ to $A_B, \ldots, A_Z$ such that $A_B$ depends solely on $B$, ..., and $A_Z$ depends solely on $Z$. Such a situation is clearly undesirable. Therefore we need to consider code duplication.

A number of different techniques have been proposed to identify the clones, among them those based on parametric string matching and metrics fingerprints. The first category of approaches extracts an abstract token stream from the code and then looks for maximal matching strings in the stream with help of a suffix tree. While these methods allow intricate duplication to be found, they sometimes can produce many insignificant results. By insignificant results we understand segments of code that match but are not necessarily cloned code. For instance, one-line code duplication is typically insignificant. The second group of approaches generates "fingerprints" by calculating a number of metrics such as maximum level of nesting, cyclomatic complexity, total number of lines, number of parameters and number of global variables for each function in the code. Functions with identical fingerprints are potential duplicates. A clear disadvantage of this approach is that it is restricted to functions as entities and hence, partially duplicated functions are unnoticed. Therefore we opted for string-matching-based technique and applied a filtering function to the results.

We used two tools for assessing code duplication, IntelliJ IDEA 4.5 and Gemini.

### 4.4.1 IntelliJ IDEA 4.5

One tool we used to locate duplicates is IntelliJ IDEA 4.5 [3]. IntelliJ IDEA is an integrated development environment supporting various development tasks such as editing, compiling, analyzing malpractices and performing refactoring. Search for duplicated code in IntelliJ IDEA starts with an abstraction step that identifies local variables, fields, literals and simple expressions visible from outside of the duplication scope. To measure simplicity of the expressions to be anonymized and to filter out some insignificant results IntelliJ IDEA applies a function, say $f$, based primarily on number of atomic expressions and atomic statements in the analyzed scope.

Code duplication turned out to be present in the case study software. 27 different clone groups have been discovered, some of them counting seven or eight instances. The longest clones appeared twice and consisted of eighteen lines of code. The highest value of $f$ is 57. We compared these results with those obtained for a content management platform InfoGlue. In InfoGlue 153 different clone groups were detected, one of them counting seventeen instances. The longest clone appeared in three files and consisted of 53 lines of code. The highest value of $f$ is 293. We have seen that the clone groups ratio (153 : 27, 5 : 66) roughly corresponds to the methods ratio (5853 : 1018, 5 : 75).

### 4.4.2 Gemini

We have also applied a special code duplication locating tool, called Gemini developed at the Osaka University [12]. Gemini is based on an earlier tool, CCFinder, which identifies code duplicates. Based on this information Gemini presents the user with a number of metrics and statistics on code duplication. We have observed that the lion share of the code duplication was found in the bsinterface package and between configuration and old_parser packages. Similarity between the files was measured by means of a similarity ratio RSA defined

<table>
<thead>
<tr>
<th>Anti-pattern</th>
<th>Importance</th>
<th>Threshold</th>
<th>Count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tangle</td>
<td>High</td>
<td>2</td>
<td>4</td>
<td>n/a</td>
</tr>
<tr>
<td>Global hub</td>
<td>High</td>
<td>10%, 10%</td>
<td>30</td>
<td>22%</td>
</tr>
<tr>
<td>Local hub</td>
<td>Medium</td>
<td>8, 8</td>
<td>11</td>
<td>8%</td>
</tr>
<tr>
<td>Global breakable</td>
<td>Medium</td>
<td>15%</td>
<td>62</td>
<td>45%</td>
</tr>
<tr>
<td>Local breakable</td>
<td>Medium</td>
<td>6</td>
<td>27</td>
<td>19%</td>
</tr>
<tr>
<td>Global butterfly</td>
<td>Low</td>
<td>15%</td>
<td>90</td>
<td>66%</td>
</tr>
<tr>
<td>Local butterfly</td>
<td>Low</td>
<td>10</td>
<td>25</td>
<td>18%</td>
</tr>
</tbody>
</table>

Table 1: Anti-patterns in the case study software
for a given file $f$ as
\[
RSA(f) = \frac{1}{LOC(f)} \sum_{c \in CF(f)} LOC(c),
\]
where $LOC(c)$ is the number of lines of code $c$, and $CF(f)$ is a set of code fragments which are included in file $f$ and have clone relation in other files. In the summation overlapping code fragments are counted only once. We have observed that the similarity ratio achieved 0.7, i.e. 70% of some files was considered as a clone of the remaining files of the system. Based on the information provided by Gemini we computed the number of duplicated lines of code, which turned out to exceed 1270, i.e., approximately 7% of the total number of lines of code.

4.4.3 Results
The tools applied agree on presence of code duplication. Code duplication ratio of 7% corresponds to the 5 to 10% code duplication in a typical large software system reported in [9]. Therefore, stability of the software with respect to code duplication issues can be estimated as average.

4.5 Implementation malpractices
To assess the quality of software, and in particular the stability, the actual implementation can not be ignored. However, most tools only analyze the structure of the code, not its behavior. For behavior analysis mainly testing and code review by humans are at hand. Testing is not very suited for assessing stability, since its results apply only to the current code, not to the code after a possible change. Human code review is very costly and for the case under study difficult, because the documentation of the code was lacking in some aspects. Hence, the code review should be supported by automatic tools. We chose to apply a tool that performs behavioral checks on the code against a formal specification. Potentially, this is very involved, since a formal specification of the code is not available and writing a full formal specification is costly and not trivial. A less ambitious approach is to check some properties that are evidently desired, but are not evidently valid. Examples of such properties are: any time a method is called, the reference to the callee is non-null; every array index is within its bounds; when a reference is cast to a subtype, the referred object is actually of that type. In general, these properties are easy to check at runtime. Unfortunately, checking these properties at runtime does not guarantee that the properties hold for all possible executions as opposed to executions corresponding to test cases. Thus, we need to apply static techniques. One can expect two types of results: either the property of interest has been formally established, or the tool has failed to achieve this. The latter can be due either to the fact that the property of interest indeed does not hold, or due to the fact that the analyzer was not intelligent enough to prove it. Because of the fine grained granularity of the checks, a failing step the analysis can focus the reviewer’s attention to a potentially problematic fragment.

A significant number of extra properties in the form of pre/postconditions of methods and class invariants have to be proved to conclude that desired properties do hold. These efforts are, in fact, very informative about the stability of the code. When it is very difficult to prove them, it can mean that correct functioning of the code is depending on a long and subtle chain of inferences that could easily be disrupted by a change in the code. For this purpose, we applied the approach of assertion checking by theorem proving, using the tool ESC/Java [2, 1]. This tool checks Java code against a specification in Java Modeling Language (JML) [8]. The subset of JML that is supported includes pre/postconditions, class invariants, loop invariants, and general assertions expressed in first-order predicate logic. ESC/Java proves these properties without intervention of the user, if it succeeds. It it fails to do so, it can only be helped by adding or changing assertions, the proving process itself cannot be influenced. ESC/Java proves properties completely modular. It proves correctness of each class in isolation, without using the implementation of other classes. Only the specification of other classes is used. This makes this approach different from model checking techniques, where a global property is checked against a model of a (sub)system that is usually larger than a single class. Furthermore, properties are directly checked against the real code, not against some model that has been created for the purpose of the assessment. Modular checking puts an extra burden on the verifier, since it requires to write specifications that are strong enough to prove correct be-
behavior of all other classes and at the same time remain true in any context the class may be used. Although more difficult, this latter consequence of modular verification is very important for quality assessment in general and stability assessment in particular. Stability implies that behavior of a class should not be compromised by a change elsewhere and this is exactly what modular verification proves. We performed only a partial verification with ESC/Java of the code under study. The reasons for this are:

1. no formal specification was available;
2. although the proving process is automated, finding the proper assertions to prove frequent properties such as absence of index errors is a time consuming task that requires much insight into the code;
3. many classes depended on classes from the Java library such as java.util.Vector, which are not completely specified.

In spite of these limitations, we found malpractices in the code that could compromise stability and are hard to find with other means than a careful code review. We stress that the malpractices found should not be considered as errors—functionality and reliability of the software are not violated. However, these malpractices can lead to unexpected results when software is modified. We give two examples.

4.5.1 Casting error reveals stability risk

In several places the method boolean equals(Object) from the class Object was reimplemented as follows:

```java
public boolean equals(Object switch) {
    return getID() == ((Switch)switch).getID();
}
```

ESC/Java produced a potential casting exception for this method, since it could be called with an argument that is not of the type Switch. In the actual code, however, it is never called this way and hence no errors will be observed in even an exhaustive test. The problem can not be solved by adding a precondition that requires the parameter switch be of the type Switch, since this precondition will be stronger than the precondition defined in the class Object for this method and hence violates the requirements of polymorphism. The only solution is to change the code (implementation or signature).

4.5.2 Correctness proof reveals immature code

A certain method contained an array index that could not be proved safe at first hand. Then it turned out that a parameter of the method was only called with the value 0, thus avoiding the indexing problem. It is clear that it could not have been the original intention of the programmers to use the method in this manner: why would they otherwise have included the parameter in the first place? The reason was that a proper treatment of other values than 0 had been postponed and never been included due to lack of time.

4.5.3 Inferred coding malpractices

From the proofs it appeared that both long chains of method calls as well as circular dependencies between classes are present in the code.

4.5.4 Results

The verdict about the code quality of the example is thus that this is rather poor. This judgment follows from the combination of directly identified malpractices like the casting error, encountered immature code and derived malpractices like long call chains and circular dependencies.

About the applicability of tooling we observe that although the assertion checking has been limited and covering only parts of the code, it revealed some serious quality problems that would be difficult to find with tools that do not take into account the behavior of the program. The casting error is a known malpractice and could be revealed by a search for code smells, but only when the search included this malpractice explicitly. The immature code problem was in fact a missing TODO comment and it will be certainly missed by tools that do not look at the actual behavior of the code. In general we conclude that assertion checking is an important complement to tools that only do a structural analysis of the code, in particular when stability is an issue.
4.6 Summary

We summarize the results in Table 2. The issues are ordered from high-level structural assessments to low-level code assessment. The overall stability assessment for the example is: “Bad code compromises good design”.

- Design is quite satisfactory
- Implementation
  - violates the design
    * flawed package communication
    * flawed architecture
  - Malpractices are introduced
- Implementation agrees with typical Software Engineering Project development practice:
  - Emphasis on early stages of development (design)
  - Lack of time and resources during the implementation.

We observe that the off-shelf tools have been successfully used for stability assessment. The interplay between different tools allows a quick and efficient estimate what can and cannot be done: e.g., if Sotograph information indicates that component structure and specification structure do not concur, it can be decided that certain further checking of properties at the ESC/Java2 level is not feasible. Also, preliminary verdicts can be refined: positive results, e.g., Sotograph’s positive judgment on functional decomposition, can be further investigated by judging coupling (“well enough”). The impact of negative results, like SA4J’s “poor” for dependency structure (tangles!) can be further assessed by looking for code malpractices (long call chains). Furthermore, results can be combined, for example to assess the development process itself (good design, but bad code is telling!).

Considering application effort of the tools an important distinction should be made between Sotograph, SA4J, IntelliJ IDEA and Gemini, on the one hand, and ESC/Java 2, on the other. First of all, applying ESC/Java 2 requires the code to be annotated while the remaining tools work on the unmodified code. Second, applying ESC/Java 2 is an iterative process: the verifier adds some annotations, the tool succeeds in proving some of them and fails in proving some other ones, which in its turn triggers the user to add new annotations and to reapply the system. All other tools are expected to be applied once. We also need to consider an effort dedicated to interpreting the results. The only tool that provides a passing threshold is SA4J (“for highly-stable systems the computed value should exceed 90%”). Understanding values of the metrics computed by Sotograph or significance of code duplication detected by IntelliJ IDEA or Gemini requires ability to compare the results obtained with benchmark software. Finally, understanding the reasons for failure of ESC/Java 2 to verify an assertion is a challenging task. We summarize the discussion above in Table 3.

<table>
<thead>
<tr>
<th>Stability issue</th>
<th>Tool</th>
<th>Information used</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional decomposition</td>
<td>Sotograph</td>
<td>Package communication diagram</td>
<td>Conform documenta-</td>
</tr>
<tr>
<td>Coupling</td>
<td>Sotograph</td>
<td>Call graph</td>
<td>Well enough</td>
</tr>
<tr>
<td>Dependency structure</td>
<td>SA4J</td>
<td>Anti-patterns (no., percentage)</td>
<td>Poor</td>
</tr>
<tr>
<td>Code duplication 1</td>
<td>IntelliJ IDEA4.5</td>
<td>Clone groups (no., instances per group)</td>
<td>Average</td>
</tr>
<tr>
<td>Code duplication 2</td>
<td>Gemini</td>
<td>Clone groups (similarity ratio)</td>
<td>Average</td>
</tr>
<tr>
<td>Implementation malpractices</td>
<td>ESC/Java2</td>
<td>Code correctness versus specification</td>
<td>Poor</td>
</tr>
</tbody>
</table>

Table 2: Assessment of the case study software
Tool | Application effort | Interpretation effort
---|---|---
Sotograph | Low | Medium
SA4J | Low | Low
IntelliJ IDEA 4.5 | Low | Medium
Gemini | Low | Medium
ESC/Java2 | High | High

Table 3: Application and interpretation effort

5 CONCLUSIONS

The code and documentation used in the case study are moderately-sized but realistic: developed by a group rather than one individual, and concerning a security layer as present in a complete system rather than an isolated protocol. Furthermore, the assessment was carried out with the LaQuSo aim to assess software in a commercially viable manner in mind rather than aiming for a specific scientific correctness result. We therefore think it justified to try to draw conclusions that, although being directly based on our experiments, extend to product software.

The first, positive, observation is, that out-of-the-box tools for quantitative assessment are available that provide support for an operationalization of the ISO definition of stability into five stability issues - moreover, support that is workable in terms of application and interpretation effort.

The second encouraging observation is, that the results that the various tools produced were consistent with one another - this applies both to the tools that assessed different but, of course, related issues as well as to the two tools that operated on the same issue, namely code duplication. Furthermore, incremental use of the tools is possible and profitable, the combination providing insights that individual tools do not supply. For example, none of the tools even addresses the development process, but good design and bad implementation suggests a process problem, like unbalanced allocation of time to these two phases.

Third, on the downside we observe that although the measurements themselves are quite clear, and consistent, it is not always clear how to interpret and weigh them. For example, more precise, quantitative, quality judgments (beyond “good”, “average”, etc.) are hard to give. Calibration against benchmarks and baselines would be useful; for some but certainly not all tools this information is present, but even then it is not always easy to judge the value. For example, it might be required to know more about other parameters: an administrative system or a process control system might score quite differently on diverse assessments but be quite similar as to stability.

Fourth, we were surprised about the positive contribution of the assertion checking tools, which we expected to be of limited value, namely only aiming for rigorously proving functional correctness rather than providing insight in stability issues. In fact the assertion checking provided two insights:

- proof complexity reflects code complexity;
- failure to prove correctness of code with respect to a specification that explicitly aims for stability, reflects lack of stability—an example is specifying applicability in a more demanding environment than the one at hand in the assessed program.

To balance these positive remarks about the assertion checking tools we remark that the lack of availability of specifications for standard APIs that are needed to do the checking is a serious limitation.

A fifth, somewhat meta-observation is, that the tooling not only provided and enforced rigor but most importantly made the assessments realistic and feasible in terms of time and effort.

Summarizing the experiences:

Tooling made the assessments feasible in terms of scale (time and size), consider for example the call graph, rigor, consider ESC/Java, and objectivity - in the hands of experts. Combinations of automated tools thus are essential for analysis, both for quality and to make assessment feasible.
Acknowledgment

We gratefully acknowledge the expert help of Erik Poll with ESC/Java2. We also like to thank Software Tomography GmbH in Munich for the provision of the Soto-
graph tool.

This paper appeared also in Proceedings of the Work-
shop on Development and Deployment of Product Soft-
ware (DDoPS-05), edited by Pradip Peter Dey, Ma-
hammad Amin, Sjaak Brinkkemper, Lai Xu (ISBN 0-

References


Dr. Cornelis Huizing holds a MSc degree in mathe-
matics (1985, Utrecht University) and received his PhD in theoretical computer science at Eindhoven University of Technology (1991), where he is currently employed. His research interests include formal and automated ver-
ication of object-oriented programs and design patterns, programming language design, and software visualization.

Dr. Ruurd Kuiper works in modular veri-
cation of programs against assertional and temporal logics speci-
cations, using model checkers and theorem provers for automation. Prior to joining the Formal Methods group at Technische Universiteit Eindhoven he was em-
ployed at the Mathematical Center Amsterdam (CWI, the Netherlands) and the Victoria University of Manchester, the United Kingdom.

Dr. Ir. Teade Punter holds a MSc degree (Ir) in Philosophy of Science and Society from Twente University and received his PhD (Dr) from Technology Management at Eindhoven University of Technology. From 1992 to 1996 he worked as course team leader for the Computer Science department at the Open University of the Netherlands in Heerlen. In 1996 he started his PhD research on software product evaluation at Eindhoven University of Technology and co-worked as a specialist for software product evaluation at Kema Nederland.
B.V. in Arnhem. From 2000 to 2004 Teade was a group leader for software product, process and subcontracting assessments at the Fraunhofer Institute for Software Engineering (Fraunhofer IESE) in Kaiserslautern, Germany. Since August 2004 he works as an ICT Consultant at LaQuSo (www.laquo.com), the Laboratory for Quality Software of Eindhoven University of Technology where he coordinates the acquisition of research projects. Teade’s interests are in quality of software design, software product assessment, software engineering for gaming and empirical software engineering.

Dr. Alexander Serebrenik obtained a Ph.D. degree in Computer Science (2003) from Katholieke Universiteit Leuven, Belgium and M.Sc. degree (1999) from the Hebrew University, Jerusalem, Israel. His areas of expertise include logic programming, termination analysis, program transformation, abstract interpretation and process modeling.
Measure to Manage Software Change

www.sig.nl

Good Cars?
That was easy... but this one?

Looks nice enough…
One lady owner, 5 years old, € 4800, 56.000km, Good condition, new roadworthy.

Software Quality

- Functionality
- Performance
- Robustness
- Reliability
- **Maintainability**
- Changeability
- ...
Software Change

• Maintenance more expensive than initial development
• Software changes continuously:
  • Functional (business demands)
  • Technical (platforms, upgrades)
• Redundancy through mergers/take-overs
• Over time complexity increases
• Software is not designed to be changeable

Design for Change

• Architectural demands:
  • Loosely coupled, modular, layered, object oriented
• Monitor:
  • Size of units
  • Dependencies (in/out)
  • Code duplication
  • Complexity/testability of code units
  • Coverage and quality of tests
  • Technical coding standards
  • Technologies in use
Software Portfolio Monitoring

- Heterogenic software systems
- Size and quality of complete portfolio
- Compare systems
- Measure changes over time
- Early indication of points of attention

Deliverables

- Annual report
- Status report
- Technical report

Diagram:
- Board of directors
- ICT Management
- Project management and developers
Technical Report

- Monthly report of factual (objective) measurements
- Automated generation of report
- Visualization through charts and graphs
- Technical detail
- Based on needs of Project Management

Examples

Complexity of sourcecode

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC McCabe 1 - 5</td>
<td>31%</td>
</tr>
<tr>
<td>LOC McCabe 6 - 10</td>
<td>22%</td>
</tr>
<tr>
<td>LOC McCabe 11 - 15</td>
<td>31%</td>
</tr>
<tr>
<td>LOC McCabe 16 - 20</td>
<td>4%</td>
</tr>
<tr>
<td>LOC McCabe 21 - 40</td>
<td>6%</td>
</tr>
<tr>
<td>LOC McCabe 40 - 60</td>
<td>2%</td>
</tr>
<tr>
<td>LOC McCabe &gt; 60</td>
<td>4%</td>
</tr>
</tbody>
</table>
**Status Report**

- Four monthly about quality of Portfolio
- Based on expert interpretation of technical reports
- Focus on trends, correlations, and exceptions in Portfolio
- Relation with business demands
- Recommendations for improvement
- Presentation followed by discussion

**Annual Report**

- State of the complete Portfolio
- Business-oriented instead of technical
- Focus on:
  - Accomplished results
  - Strategic achievements
  - Risk management
  - Next years targets
Monitoring: Process

Key values

- Automated analysis
- Suitable for heterogenic systems
- Simple to implement
- Useful in every organization
- Operative in weeks
- Usable at different levels in the organization
- Tuned to IT strategy
Measure to Manage Software Change

www.sig.nl
Since testing is inherently incomplete, test selection is of vital importance. Coverage measures evaluate the quality of a test suite and help the tester select test cases with maximal impact at minimum cost. Coverage criteria for test suites are usually defined in terms of syntactic characteristics of the implementation under test or its specification. Typical black-box coverage metrics are state and transition coverage; white-box testing often considers statement, condition and path coverage. A disadvantage of a syntactic approach is that it assigns different coverage figures to systems that are behaviorally equivalent, but syntactically different. Moreover, those coverage metrics do not take into account that certain failures are more severe than others, and that more testing effort should be devoted to uncover the most important bugs, while less critical system parts can be tested less thorough.

In this talk, I will introduce a semantic notion of test coverage for fault-based specifications. A fault-based specification gives a weight to each potential error in an implementation. We define a framework to express coverage measures that express how well a test suite covers such a specification, taking into account these error weight. Since our notions semantic, they are insensitive to replacing a specification by one that is behaviorally equivalent.

Moreover we present several algorithms that, given a certain minimality criterion, compute a minimal set suite with maximal coverage. These algorithms are based on existing and novel optimization problems.
Abstract

Let’s define the Goal of development projects as: Providing the customer with what he needs, at the time he needs it, to be more successful than he was without it, constrained by what we can deliver in a reasonable period of time. Furthermore, let’s define a defect as the cause of a problem experienced by the users of our software. If there are no problems, we will have achieved our goal. If there are problems, we failed.

We know all the stories about failed and partly failed projects. Apparently, too many defects are generated by developers, and too many remain undiscovered by checkers, causing too many problems to be experienced by users. Solutions are mostly sought in technical means like processes, metrics and tools. If this really would have helped, it should have shown by now.

Oddly enough, there is a lot of knowledge how to significantly reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in many development organizations.

In 2004, I published a booklet: How Quality is Assured by Evolutionary Methods, describing how to organize projects using this knowledge successfully. In this paper we’ll extend the use of this knowledge to testing, in order to optimize the contribution of testing to project success.

Important ingredients are: a change in attitude, taking the Goal seriously, focusing on prevention rather than repair, and constantly learning how to do things better.

1. Introduction

We know all the stories about failed and partly failed projects, only about one third of the projects delivering according to the original goal [1].

Despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by testers, causing still too many problems to be experienced by users. It seems that people are taking this state of affairs for granted, accepting it as a nature of software development. A solution is mostly sought in technical means, like process descriptions, metrics and tools. If this really would have helped, it should have shown by now.

Oddly enough, there is a lot of knowledge about how to significantly reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in the practice of many software development organizations. In papers and in actual projects I’ve observed that the time spent on testing and repairing (some people call this debugging) is quoted as being up to 60 to 80% of the total project time. That’s a large budget and provides excellent room for a lot of savings.

In an earlier paper: How Quality is Assured by Evolutionary Methods [2], I described practical implementation details of how to organize projects using this knowledge, making the project a success. In an earlier booklet: Evolutionary Project Management Methods [3], I described issues to be solved with these methods and my first practical experiences with the approach. Tom Gilb published already in 1988 about these methods [4].

In this paper we’ll extend the Evo methods to the testing process, in order to optimize the contribution of testing to project success.

Important ingredients for success are: a change in attitude, taking the Goal seriously, which includes working towards defect-free results, focusing on prevention rather than repair, and constantly learning how to do things better.
2. The goal

Let’s define as the main goal of our software development efforts: Providing the customer with what he needs, at the time he needs it, to be satisfied, and to be more successful than he was without it...

If the customer is not satisfied, he may not want to pay for our development efforts. If he is not successful, he cannot pay. If he is not more successful than he already was, why should he have invested in our product anyway?

Of course we have to add that what we do in a development project is constrained by what the customer can afford and what we mutually beneficially and satisfactorily can deliver in a reasonable period of time.

Furthermore, let’s define a defect as the cause of a problem experienced by the users of our software. Defects are caused by errors made by people. If there are no problems, we’ll have achieved our goal. If there are problems, we failed.

3. The knowledge

Important ingredients for significantly reducing the generation and proliferation of defects and delivering the right solution quicker are:

- **Clear Goal**: If we have a clear goal for our project, we can focus on achieving that goal. If management does not set the clear goal, we should set the goal ourselves.

- **Prevention attitude**: Preventing defects is more effective and efficient than injecting-finding-fixing, although it needs a specific attitude that usually doesn’t come naturally.

- **Continuous Learning**: If we organize projects in very short Plan-Do-Check-Act (PDCA) cycles, constantly selecting only the most important things to work on, we will most quickly learn what the real requirements are and how we can most effectively and efficiently realize these requirements. We spot problems quicker, allowing us more time to do something about them. Actively learning is sped up by expressly applying the Check and Act phases of PDCA.

4. Evo

Evolutionary Project Management (Evo for short) uses this knowledge to the full, combining Project-, Requirements- and Risk-Management into Result Management. The essence of Evo is actively, deliberately, rapidly and frequently going through the PDCA cycle, for the product, the project and the process, constantly reprioritizing the order of what we do based on Return on Investment (ROI), and highest value first. In my experience as project manager and as project coach, I observed that those projects, who seriously apply the Evo approach, are routinely successful on time, or earlier [5].

Evo is not only iterative (using multiple cycles) and incremental (breaking the work into smaller parts), like many similar Agile approaches, but above all Evo is about learning. We proactively anticipate problems before they occur and work to prevent them. We may not be able to prevent all the problems, but if we prevent most of them, we have a lot more time to cope with the few problems that slip through.

5. Something is not right

Satisfying the customer and making him more successful implies that the software we deliver should show no defects. So, all we have to do is delivering a result with no defects. As long as a lot of software is delivered with defects and late (which I consider a defect as well), apparently something is not right.

Customers are also to blame, because they keep paying when the software is not delivered as agreed. If they would refuse to pay, the problem could have been solved long ago. One problem here is that it often is not obvious what was agreed. However, as this is a known problem, there is no excuse if this problem is not solved within the project, well before the end of the project.

6. The problem with bugs

In a conventional software development process, people develop a lot of software with a lot of defects, which some people call bugs, and then enter the debugging phase: testers testing the software and developers trying to repair the bugs.

Bugs are so important that they are even counted. We keep a database of the number of bugs we found in previous projects to know how many bugs we should expect in the next project. Software without bugs is even considered suspect. As long as we put bugs in the center of the testing focus, there will be bugs. Bugs are normal. They are needed. What should we do if there were no bugs any more?

This way, we endorse the injection of bugs. But, does this have anything to do with our goal: making sure that the customer will not encounter any problem?

Personally, I dislike the word bug. To me, it refers to a little creature creeping into the software, causing trouble beyond our control. In reality, however, people make mistakes and thus cause defects. Using the word bug, subconsciously defers responsibility for making the mistake. In order to prevent defects, however, we have to actively take responsibility for our mistakes.
7. Defects found are symptoms

Many defects are symptoms of deeper lying problems. Defect prevention seeks to find and analyze these problems and doing something more fundamental about them.

Simply repairing the apparent defects has several drawbacks:

- Repair is usually done under pressure, so there is a high risk of imperfect repair, with unexpected side effects.
- Once a bandage has covered up the defect, we think the problem is solved and we easily forget to address the real cause. That’s a reason why so many defects are still being repeated.
- Once we find the underlying real cause, of which the defect is just a symptom, we’ll probably do a more thorough redesign, making the repair of the apparent defect redundant.

As prevention is better than cure, let’s move from fixation-to-fix to attention-to-prevention.

Many mistakes have a repetitive character, because they are a product of certain behavior or people. If we don’t deal with the root causes, we will keep making the same mistakes over and over again. Without feedback, we won’t even know. With quick feedback, we can put the repetition to a halt immediately.

8. Defects typically overlooked

We must not only test whether functions are correctly implemented as documented in the requirements, but also, a level higher, whether the requirements adequately solve the needs of the customer according to the goal. Typical defects that may be overlooked are:

- Functions that won’t be used (superfluous requirements, no Return on Investment)
- Nice things (not required, added by designers or programmers, usefulness not checked, not paid for)
- Missing quality levels (should have been in the requirements) e.g.: response time, security, maintainability, usability, learnability
- Missing constraints (should have been in the requirements)
- Unnecessary constraints (not required)
- Being late or over budget (few people learnt to treat these as defects)

Another problem that may negatively affect our goal is that many software projects end at “Hurray, it works!” If our software is supposed to make the customer more successful, our responsibility goes further: we have to make sure that the increase in success is going to happen.

This awareness will stimulate our understanding of quality requirements like “learnability” and “usability”. Without it, these requirements don’t have much meaning for development. It’s a defect if success is not going to happen.

9. Is defect free software possible?

Most people think that defect free software is impossible. This is probably caused by lack of understanding about what defect free, or Zero Defects, really means. Think of it as an asymptote (Figure 1).

We know that an asymptote never reaches its target, but we can do our best to approach the target level as closely as possible. However, if we put the bar at an acceptable level of defects, we’ll asymptotically approach that level. Only if we put the bar at zero defects, we can asymptotically approach Zero Defects.

Philip Crosby wrote [6]:

Conventional wisdom says that error is inevitable. As long as the performance standard requires it, then this self-fulfilling prophecy will come true. Most people will say: People are humans and humans make mistakes. And people do make mistakes, particularly those who do not become upset when they happen. Do people have a built-in defect ratio? Mistakes are caused by two factors: lack of knowledge and lack of attention. Lack of attention is an attitude problem.

When Crosby first started to apply Zero Defects as performance standard in 1961, the error rates dropped 40% almost immediately [6]. In my projects I’ve observed similar effects.

Zero Defects is a performance standard, set by management. In Evo projects, even if management does not provide us with this standard, we’ll assume it as a standard for the project, because we know that it will help us to conclude our project successfully in less time.
10. Attitude

As long as we are convinced that defect free software is impossible, we will keep producing defects, failing our goal. As long as we are accepting defects, we are endorsing defects. The more we talk about them, the more normal they seem. It’s a self-fulfilling prophecy. It will perpetuate the problem. So, let’s challenge the defect-cult and do something about it. From now on, we don’t want to make mistakes any more. We get upset if we make one. Feel the failure. If we don’t feel failure, we don’t learn. Then we work to find a way not to make the mistake again. If a task is finished we don’t hope it’s ok, we don’t think it’s ok, no, we’ll be sure that there are no defects and we’ll be genuinely surprised when there proves to be any defect after all.

In my experience, this attitude prevents half of the defects in the first place. Because we are humans, we can study how we operate psychologically and use this knowledge to our advantage. If we can prevent half of the defects overnight, then we have a lot of time for investing in more prevention, while still being more productive. This attitude is a crucial element of successful projects.

Experience: No more memory leaks
My first Evo project was a project where people had been working for months on software for a hand-held terminal. The developers were running in circles, adding functions they couldn’t even test, because the software crashed before they arrived at their newly added function. The project was already late and management was planning to kill the project. We got six weeks to save it. The first goal was to get stable software. After all, adding any function if it crashes within a few minutes of operation is of little use: the product cannot be sold. I told the team to take away all functionality except one very basic function and then to make it stable. The planning was to get it stable in two weeks and only then to add more functionality gradually to get a useful product.

I still had other business to finish, so I returned to the project two weeks later. I asked the team “Is it stable?” The answer was: “We found many memory leaks and solved them. Now it’s much stabler”. And they were already adding new functionality. I said: “Stop adding functionality. I want it stable, not almost stable”. One week later, all memory leaks were solved and stability was achieved. This was a bit of a weird experience for the team: the software didn’t crash any more. Actually, in this system there was not even a need for dynamically allocatable memory and the whole problem could have been avoided. But changing this architectural decision wasn’t a viable option at this stage any more.

Now that the system was stable, they started adding more functions. We got another six weeks to complete the product. I made it very clear that I didn’t want to see any more memory leaks. Actually that I didn’t want to see any defects. The result was that the testers suddenly found hardly any defect any more and from now on could check the correct functioning of the device. At the end of the second phase of six weeks, the project was successfully closed. The product manager was happy with the result.

Conclusion: after I made it clear that I didn’t want to see any defects, the team hardly produced any defects. The few defects found were easy to trace and repair. The change of attitude saved a lot of defects and a lot of time. The team could spend most of its time adding new functionality instead of fixing defects. This was Zero Defects at work. Technical knowledge was not the problem to these people: once challenged, they quickly came up with tooling to analyze the problem and solve it. The attitude was what made the difference.

Experience: No defects in the first two weeks of use
A QA person of a large banking and insurance company I met in a SPIN metrics working group told me that they got a new manager who told them that from now on she expected that any software delivered to the (internal) users would run defect free for at least the first two weeks of use. He told me this as if it were a good joke. I replied that I thought he finally got a good manager, setting them a clear requirement: “No defects in the first two weeks of use.” Apparently this was a target they had never contemplated before, nor achieved. Now they could focus on how to achieve defect free software, instead of counting function points and defects. Remember that in bookkeeping being one cent off is already a capital offense, so defect free software should be a normal expectation for a bank. Why wouldn’t it be for any environment?
11. Plan-Do-Check-Act

I assume the Plan-Do-Check-Act (PCDA- or Deming-) cycle [7] is well known (Figure 2).

Because it’s such a crucial ingredient, I’ll shortly reiterate the basic idea:
- We Plan what we want to accomplish and how we think to accomplish it best.
- We Do according to the plan.
- We Check to observe whether the result from the Do is according to then Plan.
- We Act on our findings. If the result was good: what can we do better. If the result was not so good: how can we make it better. Act produces a renewed strategy.

The key-ingredients are: planning before doing, systematically checking and above all acting: doing something differently. After all, if you don’t do things differently, you shouldn’t expect a change in result, let alone an improvement in result.

In Evo we constantly go through multiple PDCA cycles, deliberately adapting strategies, in order to learn how to do things better all the time, actively and purposely speeding up the evolution of our knowledge. As a driver for moving the evolution in the right direction, we use Return on Investment (ROI): the project invests time and other resources and this investment has to be regained in whatever way, otherwise it’s just a hobby. So, we’ll have to constantly be aware whether all our actions contribute to the value of the result. Anything that does not contribute value, we shouldn’t do.

Furthermore, in order to maximize the ROI, we have to do the most important things first. In practice, priorities change dynamically during the course of the project, so we constantly reprioritize, based on what we learnt so far. Every week we ask ourselves: “What are the most important things to do. We shouldn’t work on anything less important.” Note that priority is molded by many issues: customer issues, project issues, technical issues, people issues, political issues and many other issues.

12. How about Project Evaluations

Project Evaluations (also called Project Retrospectives, or Post-Mortems - as if all projects die) are based on the PDCA cycle as well. At the end of a project we evaluate what went wrong and what went right.

Doing this only at the end of a project has several drawbacks:
- We tend to forget what went wrong, especially if it was a long time ago.
- We put the results of the evaluation in a write-only memory: do we really remember to check the evaluation report at the very moment we need the analysis in the next project? Note that this is typically one full project duration after the fact. So there is not much benefit for the next project.
- The evaluations are of no use for the project just finished and being evaluated.
- Because people feel these drawbacks, they tend to postpone or forget to evaluate. After all, they are already busy with the next project, after the delay of the previous project.

In short: the principle is good, but the implementation is not tuned to the human time-constant.

In Evo, we evaluate weekly (in reality it gradually becomes a way-of-life), using PDCA cycles, and now this starts to bear fruit (Figure 3):
- Not so much happens in one week, so there is not so much to evaluate.
- It’s more likely that we remember the issues of the past five days.
- Because we most likely will be working on the same kind of things during the following week, we can immediately use the new strategy, based on our analysis.
- One week later we can check whether our new strategy was better or not, and refine.
- Because we immediately apply the new strategy, it naturally is becoming our new way of working.
- The current project benefits immediately from what we found and improved.
Evaluations are good, but they must be tuned to
the right cycle time to make them really useful. The
same applies to testing, as this is also a type of
evaluation.

13. Current Evo testing

In conventional development mode, most verifi-
cation is still executed in Waterfall mode: developers
are first allowed to inject defects (in drawings,
designs, or pieces of code), then testers and checkers
are supposed to find the defects injected, after which
the developers are supposed to repair the defects
found. In reality, testers and checkers find only part
(30 – 80%) of the defects injected (testers and
checkers are human as well). In Evo, we humbly
admit that we probably don’t know the real require-
ments, that we have to check our assumptions and
that we are prone to making mistakes. Evo testers
assist the development people to reach their goal
successfully. This includes verification of all phases
of the development process and ploughing back the
findings to the developers for optimizing the prod-
uct, the project and the process.

Developers design the order of Deliveries in such
a way that, in case they made an erroneous assump-
tion or a downright error, it will be found as quickly
as possible (Figure 4). This way, most of any undis-
covered defects will be caught before the final
delivery and, more importantly, be exploited for
prevention of further injection of similar defects.
Evo projects do not need a separate verification
(sometimes called “debugging”) phase and hardly
need repair after delivery. If a delivery is ready, it is
complete. Anything is only ready if it is completely
done, not to worry about it any more. That includes:
no defects. I know we are human and not perfect, but
remember the importance of attitude: we want to be
perfect. Because all people in the project aim for
Zero Defects delivery, the developers and testers
work together in their quest for perfection.

Note that perfection means: freedom from fault or
defect. It does not mean: gold plating.

14. Further improvement

In the original Evo concept we gained a lot by
preventing the injection of defects, because people
learn during the work: if a designer has to produce
several documents (plans, drawings, designs, pieces
of code) or pieces of hardware, he can learn from his
mistakes made in the first item, and avoid making
similar mistakes in subsequent similar work.

As long as single pieces of work are still made in
waterfall mode, first “completed” and only subse-
quently checked, we are still waiting for the
designers to inject defects first, hoping that we can
find and fix all these defects afterwards. In order to
drive prevention further, why don’t we contemplate
checking the result of designers before the first item
is completed, so that they can prevent mistakes
immediately, avoiding the waterfall-syndrome even
on single pieces of work. This may seem overkill in
case of a first small document of a large set of
documents. It makes a lot of sense, however, in case
of one single, or a relatively large document.

We can also extend the Evo project management
techniques to the QA process itself and exploit the
PDCA paradigm even further:

• Testers focus on a clear goal. Finding defects is
not the goal. After all, we don’t want defects. Any
defects found are only a means to achieve
the real goal: the success of the project.

• Testers will select and use any method appropri-
ate for optimum feedback to development, be it
testing, review or inspection, or whatever more
they come up with.

• Testers check work in progress even before it is
delivered, to feedback issues found, allowing the
developer to abstain from further producing these
issues for the remainder of his work.

“Can I check some piece of what you are work-
ing on now?” “But I’m not yet ready!” “Doesn’t
matter. Give me what you have. I’ll tell you what
I find, if I find anything”. Testers have a different
view, seeing things the developer doesn’t see.
Developers don’t naturally volunteer to have
their intermediate work checked. Not because
they don’t like it to be checked, but because their
attention is elsewhere. Testers can help by ask-
ing. Initially the developers may seem a little

![Figure 4: Testing of early deliveries helps the developers to get ready for zero-defect final delivery](image)
surprised, but this will soon fade. If the testers play this game well.

- Similarly, testers can solve a typical problem with planning reviews and inspections. Developers are not against reviews and inspections, because they very well understand the value. They have trouble, however, planning them in between of their design work, which consumes their attention more. If we include the testers in the process, the testers will recognize when which types of review, inspections or tests are needed and organize these accordingly. This is a natural part of their work helping the developers to minimize rework by minimizing the injection of defects and minimizing the time slipped defects stay in the system.

In general: organizing testing the Evo way means entangling the testing process more intimately with the development process.

15. Cycles in Evo

In the Evo development process, we use several learning cycles (see [2] and [3] for explanations of terms):
- The TaskCycle [9] is used for organizing the work, optimizing estimation, planning and tracking. We constantly check whether we are doing the right things in the right order to the right level of detail. We optimize the work effectiveness and efficiency. TaskCycles never take more than one week.
- The DeliveryCycle [10] is used for optimizing the requirements and checking the assumptions. We constantly check whether we are moving to the right product results. DeliveryCycles focus the work organized in TaskCycles. DeliveryCycles normally take not more than two weeks.
- TimeLine [11] is used to keep control over the project duration. We optimize the order of DeliveryCycles in such a way that we approach the product result in the shortest time, with as little rework as possible.

During these cycles we are constantly optimizing:
- **The product** [12]: how to arrive at the best product (according to the goal).
- **The project** [13]: how to arrive at this product most effectively and efficiently.
- **The process** [14]: finding ways to do it even better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

If we do this well, by definition, there is no better way.

16. Evo cycles for testing

Extending Evo to testing adds cycles (Figure 5) for feedback from testing to development, as well as cycles for organizing and optimizing the testing activities themselves:

- Testers organize their work in weekly, or even shorter TaskCycles.
- The DeliveryCycle of the testers is the Test-feedback cycle: in very short cycles testers take intermediate results from developers, check for defects in all varieties and feed back optimizing information to the developers, while the developers are still working on the same results. This way the developers can avoid injecting defects in the remainder of their work, while immediately checking out their prevention ideas in reality.
- The Testers use their own TimeLine, synchronized with the development TimeLine, to control that they plan the right things at the right time, in the right order, to the right level of detail during the course of the project and that they conclude their work in sync with development.

During these cycles the testers are constantly optimizing:
- **The product**: how to arrive at the most effective product. Remember that their product goal is: providing their customer, in this case the developers, with what they need, at the time they need it, to be satisfied, and to be more successful than they were without it.
- **The project**: how to arrive at this product most effectively and efficiently.
- This is optimizing in which order they should do which activities to arrive most efficiently at their result.
- **The process**: finding ways to do it better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.
Testers are part of the project and participate in the weekly 3-step procedure [15] using about 20 minutes per step:
1. Individual preparation.
2. 1-to-1’s: Modulation with and coaching by Project Management.
3. Team meeting: Synchronization and synergy with the team.

Project Management in step 2 of the 3-step procedure is now any combination, as appropriate, of the following functions:
- The Project Manager/Leader, for the project issues.
- The Architect, for the product issues.
- The Test Manager, for the testing issues.

There can be only one captain on the ship, so the final word is to the person who acts as Project Manager, although he should better listen to the advice of the others.

Testers participate in requirements discussions. They communicate with developers in the unplannable time [16], or if more time is needed, they plan tasks for interaction with developers. If the priority of an issue is too high to wait for the next TaskCycle, the interrupt procedure [17] will be used. If something is unclear, an Analysis Task [18] will be planned. The Prevention Potential of issues found is an important factor in the prioritization process.

In the team meeting testers see what the developers will be working on in the coming week and they synchronize with that work. There is no ambiguity any more about which requirements can be tested and to which degree, because the testers follow development, and they design their contribution to assist the project optimally for success.

In Evo Testing, we don’t wait until something is thrown at us. We actively take responsibility. Prevention doesn’t mean sitting waiting for the developers. It means to decide with the developers how to work towards the defect free result together. Developers doing a small step. Testers checking the result and feeding back any imperfections before more imperfections are generated, closing the very short feedback loop. Developers and testers quickly finding a way of optimizing their cooperation. It’s important for the whole team to keep helping each other to remind that we don’t want to repair defects, because repair costs more. If there are no defects, we don’t have to repair them.

In many cases, the deadline of a project is defined by genuine external factors like a finite market-window. Then we have to predict which requirements we can realize before the deadline or “FatalDate”. Therefore, we still need to estimate the amount of work needed for the various requirements. We use the TimeLine technique to regularly predict what we will have accomplished at the FatalDate and what not, and to control that we will have a working product well before that date. Testers use TimeLine to control that they will complete whatever they have to do in the project, in sync with the developers.

Doesn’t all of this take a lot of time? No. My experience with many projects shows that it saves time, projects successfully finishing well before expected. At the start it takes some more time. The attitude, however, results in less defects and as soon as we focus on prevention rather than continuous injection-finding-fixing, we soon decrease the number of injected defects considerably and we don’t waste time on all those defects any more.

17. RI/CR/PR database

Most projects already use some form of database to collect defects reported (PR/Problem Report: development pays) and proposed changes in requirements (CR/Change Request: customer pays).

If we are seriously in Prevention Mode, striving for Zero Defects, we should also collect Risk Issues (RI): issues which better be resolved before culminating into CR’s or PR’s.

With the emphasis shifted from repair to prevention, this database will, for every RI/CR/PR, have to provide additional space for the collection of data to specifically support the prevention process, like:
- Follow-up status.
- When and where found.
- Where caused and root cause
- Why didn’t we find it earlier
- Prevention plan
- Analysis task defined and put on the Candidate Tasks List [19].
- Prevention task(s) defined and put on the Candidate Tasks List.
- Check lists updated for finding this issue easier, in case prevention doesn’t work yet.

Analysis tasks may be needed to sort out the details. The analysis, prevention and repair tasks are put on the Candidate Tasks List and will, like all other candidate tasks, be handled when their time has come: if nothing else is more important. Analysis tasks, prevention tasks and repair tasks should be separated, because analysis and prevention usually have priority over repair. We better first stop the leak, to make sure that not more of the same type of defect is injected.

18. How about metrics?

In Evo, the time to complete a task is estimated as a TimeBox [20], within which the task will be 100% done. This eliminates the need for tracking consid-
erably. The estimate is used during the execution of the task to make sure that we complete the task on time. We experienced that people can quite well estimate the time needed for tasks, if we are really serious about time.

Note that exact task estimates are not required. Planning at least 4 tasks in a week allows some estimates to be a bit optimistic and some to be a bit pessimistic. All we want is that, at the end of the week, people have finished what they promised. As long as the average estimation is OK, all tasks can be finished at the end of the week. As soon as people learn not to overrun their (average) estimates any more, there is no need to track or record overrun metrics. The attitude replaces the need for the metric. So, we do use metrics and measurements in Evo, but more, there is no need to track or record overrun metrics. The attitude replaces the need for the metric. Instead of improving non-value adding activities, including various types of metrics, it is better to eliminate them. In many cases (but not all!), the attitude, and the use of the Evo techniques replace the need for metrics. Other metrics may still be useful, like Remaining Defects, as this metric provides information about the effectiveness of the prevention process. Still, even more than in conventional metrics activities, we will be on the alert that whatever we do must contribute value.

If people have trouble deciding what the most important work for the next week is, I usually suggest as a metric: “The size of the smile on the face of the customer”. If one solution does not get a smile on his face, another solution does cause a smile and a third solution is expected to put a big smile on his face, which solution shall we choose? This proves to be an important Evo metric that helps the team to focus.

19. Finally

Many software development organizations in the world are working the same way, producing defects and then trying to find and fix the defects found, waiting for the customer to experience the reminder. In some cases, the service organization is the profit-generator of the company. And isn’t the testing department assuring the quality of our products? That’s what the car and electronics manufacturers thought until the Japanese products proved them
wrong. So, eventually the question will be: can we afford it?

Moore’s Law is still valid, implying that the complexity of our systems is growing exponentially, and the capacity needed to fill these systems with meaningful software is growing exponentially even faster with it. So, why not better become more productive by not injecting the vast majority of defects. Then we have more time to spend on more challenging activities than finding and fixing defects. I absolutely don’t want to imply that finding and fixing is not challenging. Prevention is just cheaper. And, testers, fear not: even if we start aiming at defect free software, we’ll still have a lot to learn from the mistakes we’ll still be making.

Dijkstra [8] said:

*It is a usual technique to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*

Where we first pursued the very effective way to show the presence of bugs, testing will now have to find a solution for the hopeless inadequacy of showing their absence. That is a challenge as well.

I invite testers from now on to change their focus from finding defects, to working with the developers to minimize the generation of defects in order to satisfy the real goal of software development projects. Experience in many projects shows that this is not an utopia, but that it can readily be achieved, using the Evo techniques described.

**References**


[5] See cases: [http://www.malotaux.nl/nrm/Evo/EvoFCases.htm](http://www.malotaux.nl/nrm/Evo/EvoFCases.htm)


Modern software designs are increasingly *asynchronous* and *concurrent*. Such systems are, by definition, *nondeterministic*, increasingly complex and introduce the potential for design errors such as deadlocks, divergence and race conditions. These are among the most difficult errors to detect and remove by testing. It is axiomatic that *nondeterministic systems are untestable*. There is no economically feasible amount of testing that can give us any meaningful measures of correctness and freedom from errors.

All testing is an exercise in sampling, but in testing software systems, the sample size is very small compared to the population size. Consider a simple software module with an alphabet of 20 stimuli and a maximum sequence length of 10 (that is, the longest sequence of input stimuli that results in unique behaviour). There are in the order of $1.08E13$ potential execution scenarios. Now imagine two different components of this complexity executing concurrently and communicating on a shared an alphabet of 10 events. How many potential execution scenarios are there? Now imagine compositions of 20 such processes, or a 100 or more. How can conventional, informal design methods address such complexity? What does testing coverage mean in this context?

Software engineering differs from all other branches of engineering in one important way; all other branches of engineering routinely use appropriate branches of mathematics to verify specifications and design before construction. Software engineering uniquely does not. Most software is developed *without* the use of mathematics during specification and design. Designs cannot be verified before construction. Testing must therefore test specifications, designs *and* implementation. Testing is the least certain, most expensive way to detect and remove specification and design errors because it occurs *after* the software has been implemented and because the systems we design are nondeterministic. Testing also occurs at a time when defect detection and repair has the greatest impact of time-to-market.

Improving testing methods and tools will result in limited improvement in testing costs and effectiveness; the greatest gain is to be made by reducing the errors in the software when it enters testing. How can we do this?
Verum Design Principles

- Business Critical Software must be based on designs that are verifiably correct before implementation

- Software Architects and Designers must limit themselves to designs that can be verified

We can follow the routine practices of other branches of engineering. Verum designs and develops *Business Critical* for its Clients; that is software essential to some core product or service our Clients provide to their customers. Predictable cost, quality and time-to-market are key issues for our Clients. These are the two golden rules that govern how we meet these requirements. The rest of this presentation addresses how we put the first principle into practice.
For new software, either for new systems or for new parts of existing systems, we start with a conventional "informal" specification in the form of the work products already produced by our Customer's existing development process. Step 1 is to make an ASD specification using Sequence-based Specification techniques (SBS) to produce a so-called Back Box Function (BB) specifying the required functional behaviour. This is a total mathematical function mapping all possible sequences of input stimuli (events, messages method calls etc.) onto the specified system response. We do this together with Customer domain / technical experts. The goal here is precision, not detail as such.

For reengineering existing software components either because of required changes or because conventional testing based approaches have been unable to solve stability or reliability problems, we may also reverse engineer the specifications from the existing code base, again with the involvement as needed from those familiar with the code.

When we have completed the ASD specification, we must establish 1) that it matches the original specification 2) that the design fully implements it and 3) that the code fully implements the design.

The first we do by inspection. This is possible because although the ASD specifications are based on mathematical principles, they do not use difficult mathematical notations. They are easily accessible to stakeholders and fully traceable to the original specifications. The other questions are answered next – starting with the design.
We make the design following generally accepted, conventional approaches, the big differences being 1) the emphasis we place on precision and 2) the way in which we document the design. Function behaviour is captured using SBS in the form of a design BB. Again, the ASD specifications allow full participation of other engineers because they do not rely on much visible mathematics. Most software engineers learn this technique quite quickly and like it.

If we are reengineering and existing component, then during the design we may reverse engineer much of the design from the existing code.
Having done this, we have a “proof” obligation to discharge; namely verifying the BB function of the design against the BB we made from the requirements. How do we know the design implements everything in the requirements and nothing else? How do we know it will behave according to its functional requirements?
We translate the BB specifications of the requirements and the design automatically to CSP models and we use the model checker FDR to establish that the BB design exactly complies with it. The way we apply SBS to specifications enables nondeterminism to be captured properly, essential when describing externally visible behaviour. CSP algebra also captures nondeterminism and the refinement principles used in CSP are able to compare deterministic design models mathematically to nondeterministic specification models. The mathematical verification we use in this case is called Failures Refinement. With this, we can verify whether or not the design (i) specifies all required behaviour in the correct way; (ii) does not specify any behaviour not specified in the specification and (iii) if optional behaviour is specified in the design, it is designed according to the specification. These are not inspections or tests; these are mathematical proofs so they hold for all possible execution scenarios. We could never establish this by testing.
But of course, in reality, we cannot establish that a design behaves correctly without considering how it interacts with the other components it uses. Indeed, the way in which the design will interact with other components, HW or SW, is a key part of establishing that the design is correct. Particularly in event driven, reactive systems with concurrent behaviour, this cannot be done by inspecting static design specifications individually. We need some way of exploring the dynamic behaviour of the design as it will behave together with its runtime environment when it executes. And of course, we wish to do this before we implement our designs in code. How do we do this?
We apply SBS to analyse the externally visible behaviour of these other components and make BB function specifications of them. This is a valuable exercise in itself; it leads to a more complete and deeper understanding of the behaviour of these other components; it focuses on interface behaviour and frequently raises important questions not clearly addressed in the conventional interface specifications. It looks like new work, but it is not; we have to do this analysis and understanding anyway in order to successfully program against these interfaces even in a conventional development process. The new work is just capturing this knowledge as a BB function and we get a huge payoff for this little extra effort. We verify this work by inspection and discussion with "experts".

When implementing new software components that are to be a part of an existing legacy system, it is frequently the case that the current implementation of the legacy software no longer behaves according to the existing specifications. In these situations, the ASD specifications will be made with frequent reference to the existing legacy code base, "recovering" the current specifications from the existing implementation.

Having done this, we generate the CSP models of these interfaces and check our design together with these interface models.
At this point, we have a design which is verified against the functional requirements. We now have to implement this and verify the implementation against the design. The BB specification of the design is not a good programming specification – it uses abstractions such as infinite sequences of abstract events that are difficult to represent in most programming languages. The “abstraction” step is too big to expect a programmer to move directly from the BB specification to code. These abstractions have to be made more concrete before we can program them. This is done using the Box Structured Development Method (BSDM). This gives us a mathematically sound way to transform the BB into a State Box (SB) in which all these difficult abstractions are replaced by state data and state data update rules. We can program directly from this and we can check the code against this by inspection.

But first, we must establish that we made no mistakes and the SB exactly refines the BB.
This we do by automatically generating the corresponding CSP model of the SB and using a mathematical refinement called traces refinement to establish that the SB describes exactly the same behaviour as the BB. This is checked using the model checker.

We address the issue of programming compliance with the design in three ways:

1. Some code (it depends on each project as to how much) can be generated automatically and we do not need to check this at all;

2. Some code still has to be hand written and checking this against verified designs in the form of SB specifications is straightforward using inspection;

3. We can generate large numbers of test cases in the form of self running test programs, execute the tests and analyse the results automatically. This testing is based on statistical concepts and is very cost efficient and effective.

By applying these techniques in this manner, components should enter integration testing with far fewer defects than is usual. Also, because we are able to analyse dynamic behaviour between components before investing in programming, there should be far fewer difficult integration defects to detect and remove.
This gives us a number of important advantages.

(i) We can verify specifications and designs before we invest in implementation. This is both cheaper and more certain than testing; it is also much quicker.

(ii) We can analyse the dynamic behaviour of designs before implementation; including behaviour between components as well as within individual components. Because models are generated automatically, we don’t need to verify models against specifications and we have no traceability issues.

(iii) In safety critical areas, we can work with domain safety engineers to analyse safety cases and formulate them as safety specifications to be verified by refinement. This means we can verify designs mathematically and ensure that such safety case hold. Again, this is not inspection or testing, but mathematical proof, providing a degree of certainty not achievable any other way.

(iv) Most importantly, ASD can be added to existing project teams in existing environments with minimum disruption and stakeholders retain control over specifications because they can understand and verify ASD specifications.
ASD Benefits

- Software enters testing with 90% fewer defects
  - Conventional testing more effective
  - Testing becomes quality control instead of quality assurance
  - Testing can concentrate on aspects we cannot verify mathematically and complement the development process
  - Fewer defects reach end users
  - Actual and perceived quality much higher
- Development costs reduced by 30% or more
  - Less Rework
  - Removal of many defects early in the lifecycle means much less unpredictable corrective rework later.
- Development time reduced by 30% or more
  - Shorter Time-to-Market
  - Fewer defects means shorter testing cycles & less rework
- Improved Predictability
  - In terms of cost, time to market and quality

This is the connection to the “bottom line” business goals of the organisation. This is our experience and that of our Customers based on the projects we have completed so far. Software development by ASD is cheaper, quicker and results in fewer defects reaching end users.

All of this translates to bottom line profit increase and competitive advantage.

Because software enters testing with far fewer specification and design errors, testing can concentrate on detecting construction errors and those defects that we cannot easily verify mathematically.

Because we have eliminated the difficult, nondeterministic design errors such as deadlocks and race conditions before construction, the errors that remain will be more easily reproducible, quicker to detect by testing and quicker and cheaper to repair.
Eindhoven, 24 November 2005

Gartner

- The Gartner Company
- The CIO agenda 2005
- Performance management
- The Benchmarking principles and approach
- World wide metrics
- World Class Performance
The Gartner Businesses

- **Gartner Research**
  - Gartner is the premier source of objective, independent intelligence on information technology.

- **Events**
  - Gartner worldwide events such as symposia and conferences give clients live access to insights developed from our research in a very concentrated way. Gartner Symposium/ITxpo® is the largest and most strategic conference for senior IT and business professionals.

- **Gartner Executive Programs**
  - Building on the foundation provided by Gartner research, we offer programs combining research with networking and advisory opportunities for chief information officers (CIOs), their direct reports and other senior business executives.

- **Gartner Consulting**
  - Gartner provides customized project consulting and strategic advice to CIOs and other senior business executives. Our consulting services are provided by 600+ consultants and focus on selected areas that are critical to clients today.
Capabilities of Gartner Consulting

- **Gartner is Business-Focused.**
  - The integration of results-oriented business strategy with all the power of Gartner Research.

- **Gartner Provides Direct Access to Analysts.**
  - Gartner consultants are working with Gartner analysts every day, well ahead of the information curve.

- **Gartner Ensures Assured Accuracy.**
  - Gartner consultants work exclusively with Gartner tools and methodologies, including the largest and most accurate peer databases available.

- **Gartner is an Independent Partner You Can Trust.**
  - Gartner’s independence is critically important. Gartner remains resolutely objective.

- **Gartner Delivers End-to-End Life Cycle Support.**
  - From opportunity to measurement to strategy to real results, Gartner Consulting can support you through the entire business and technology life cycle.

Gartner's CIO agenda 2005 results

<table>
<thead>
<tr>
<th>To what extent is each of the following CIO actions a priority for you in 2005?</th>
<th>Rank 2005</th>
<th>Rank 2004</th>
<th>Rank 2003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivering projects that enable business growth</td>
<td>1 △</td>
<td>18 ▲</td>
<td>**</td>
</tr>
<tr>
<td>Linking business and IT strategies and plans</td>
<td>2 △</td>
<td>4 ▲</td>
<td>6</td>
</tr>
<tr>
<td>Demonstrating the business value of IS/IT</td>
<td>3 ▼</td>
<td>2 ▲</td>
<td>2</td>
</tr>
<tr>
<td>Applying metrics to IS organization and services</td>
<td>4 △</td>
<td>14 ▲</td>
<td>**</td>
</tr>
<tr>
<td>Tightening security and privacy safeguards</td>
<td>5 △</td>
<td>6 ▲</td>
<td>10</td>
</tr>
<tr>
<td>Improving business continuity readiness</td>
<td>6 △</td>
<td>12 ▲</td>
<td>**</td>
</tr>
<tr>
<td>Improving the quality of IS service delivery</td>
<td>7 ▼</td>
<td>1 ▲</td>
<td>8</td>
</tr>
<tr>
<td>Consolidating the IS organization and operations</td>
<td>8 ▼</td>
<td>3 ▲</td>
<td>**</td>
</tr>
<tr>
<td>Developing leadership in the senior IS team</td>
<td>9 ▲</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Improving IT governance</td>
<td>10 ▲</td>
<td>11 ▲</td>
<td>3</td>
</tr>
</tbody>
</table>

△▼ Selected change in ranking compared with 2004
* New question for 2005
** New question in 2004
Next prominent needs require comprehensive performance management

- Delivering projects that enable business growth
- Linking business and IT strategies and plans
- Demonstrating the business value of IS/IT
- Applying metrics to IS organization and services
- Improving the quality of IS service delivery

The Performance management challenges
- The Benchmarking principles and approach
- The Benchmark Process
For performance optimization; Determine the current level of IT service maturity and a three-year planning

Organizations pursue process optimization through various initiatives around high priority disciplines

Performance optimization maps out the journey to exploiting the investment in IT

Performance Optimization Solutions

Process Reengineering → Service Management
Benchmarking Solutions
There is a danger of not properly selecting the right performance optimization objectives

- Too low in ambition
- Optimizing the inward focussed IT operations and losing the competition
- Missing customer focussed and service management opportunities
- Overlooking options to improve competencies and processes
- Not being innovative and pro-active on changes in the market
- Losing responsiveness
- Etc.

The typical Benchmark challenges

- Manage performance
  "We use benchmarking inside our performance management programs to track our performance and to identify areas that are performing poorly."

- Reduce costs
  "When we work to identify any significant IT cost-reduction opportunities and set target cost levels — we use benchmarking."

- Find working capital
  "The results of our benchmarking studies often help us identify where we can increase efficiencies in areas that are in maintenance mode to fund capital investments."

- Recommend change
  "We have typically used benchmarking whenever we have made investments in infrastructure or as we make major work process changes to justify our recommendations."
Varying degrees of Benchmarking provide varying levels of Insight

- **Investment**: Time, Resources, Cost
- **Decision**: Identifies how the benefits can be realized • Provides a detailed assessment focusing on both cost and service • Offers recommendations for delivering greater cost-efficiency of the IT function and improved service delivery
- **Status**: Identifies IT staffing and spending and compares them with those of other enterprises • Provides clear insights into the potential for improved cost-efficiency and observations targeting where the benefits may be realized
- **Analysis**: Depth, Complexity, Recommendations
- **Optimization**: Captures a holistic view of the IT function and assesses how best to optimize total functionality • A multidimensional analysis that may include any or all of the following: cost-efficiency assessment, service delivery assessment, application portfolio analysis, process review, alignment assessment and governance review

What information or metrics does a benchmark normally provide?

- IT as percentage of sales or revenue
- IT investment as percentage of revenue
- IT price or value related to the business process output
- IT price related to the required service levels
- IT cost related to the delivered service levels
- IT cost related to the total cost of ownership (TCO)
- IT operating budget per employee
- Number on FTE required versus the number on FTE consumed
- The salaries per FTE in the market versus the salaries per FTE paid.
What are the key metrics?

- Using high level metrics only doesn't help to understand the IT performance
- High level metrics don't reveal the necessary details
- High level metrics can lead to false conclusions and therefore to ineffective performance optimization objectives

- For successful decisions in performance optimization, one must have bottom-up supported and fact based metrics!

The benchmark cycle

Step 1: The service catalogue is in the middle

Step 8: Compare the performance with all industries IT metrics

Step 7: Define the metrics per activity

Step 6: Combine all activities into an overall performance overview

Step 5: Report the results per activity

Step 2: Define the scope of interest

Step 3: Decompose the IT services in scope into activities

Step 4: Analyse each activity in scope
Step 1: The service catalogue is in the middle

Service Receiver

Allocation of price to the used services

- Basic infrastructure services
- Desktop services
- Applications services

Service catalogue

- Hardware
- Software
- Transmission
- Occupancy
- Dis. recovery
- FTE costs

Service Provider

Allocate to main services categories

Allocate to main cost categories

All IT Costs From account systems, time writing etc.

Unit of Measurement:
- Used volume
- Price

Step 2: Define the scope of interest

Board

Central functions

HR

Fin

AM

Staff

Proc

Arch

Example: Organisational scoping

- Out scope
- In scope, indirect FTE
- In scope, (in)direct FTE
Step 3: Decompose the IT services in scope into activities

<table>
<thead>
<tr>
<th>Prices</th>
<th>Services</th>
<th>Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monthly fee for the use of the infrastructure and hardware</td>
<td>Basic Infrastructure services</td>
<td>LAN</td>
</tr>
<tr>
<td>Variable price for the different usage of the service packages</td>
<td>Desktop services</td>
<td>WAN</td>
</tr>
<tr>
<td>Variable price for the different usage of the application packages</td>
<td>Applications services</td>
<td>Infra structure servers</td>
</tr>
</tbody>
</table>

Step 4: Analyse each activity in scope

Complexity = function (service level, quantities, objects, ..)

- Distributed Computing (# users)
- ICT Helpdesk (# calls)
- Midrange servers (# servers)
- Widearea data (# devices)
- Applications development (# functionpoints)
- Applications support (# functionpoints)

Effectiveness = result_measured / result_norm *100%

Efficiency = offers_norm / offers_measured * 100%

Productivity = result / offers
Step 5: Report the results per activity

<table>
<thead>
<tr>
<th>Cost in €1000 per year</th>
<th>Target</th>
<th>Saving potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developing</td>
<td>Average</td>
<td>Mature</td>
</tr>
<tr>
<td>Occupancy</td>
<td>4,405</td>
<td>1,472</td>
</tr>
<tr>
<td>Hardware</td>
<td>5,428</td>
<td>4,682</td>
</tr>
<tr>
<td>Software</td>
<td>3,800</td>
<td>1,646</td>
</tr>
<tr>
<td>Personnel direct</td>
<td>21,831</td>
<td>21,831</td>
</tr>
<tr>
<td>Personnel indirect</td>
<td>3,001</td>
<td>50,627</td>
</tr>
<tr>
<td>Personnel external</td>
<td>24,865</td>
<td>24,865</td>
</tr>
<tr>
<td>Outsourcer</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>Total Cost</td>
<td>64,259</td>
<td>56,079</td>
</tr>
<tr>
<td>TCO metric (cost per dev FP/year in €)</td>
<td>194</td>
<td>170</td>
</tr>
</tbody>
</table>

Step 6: Combine all activities into an overall performance overview

<table>
<thead>
<tr>
<th>Gartner database</th>
<th>2003</th>
<th>2004</th>
<th>developing</th>
<th>Average</th>
<th>Mature</th>
<th>Advice</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD cobol</td>
<td>28,500</td>
<td>22,000</td>
<td>35,000</td>
<td>22,000</td>
<td>22,000</td>
<td>3,000</td>
<td></td>
</tr>
<tr>
<td>AS cobol</td>
<td>4,300</td>
<td>4,800</td>
<td>10,400</td>
<td>8,000</td>
<td>5,000</td>
<td>4,800</td>
<td></td>
</tr>
<tr>
<td>Mainframe</td>
<td>28,300</td>
<td>20,500</td>
<td>31,000</td>
<td>24,200</td>
<td>17,300</td>
<td>17,300</td>
<td></td>
</tr>
<tr>
<td>AD Java</td>
<td>59,100</td>
<td>50,300</td>
<td>86,800</td>
<td>67,200</td>
<td>44,100</td>
<td>44,100</td>
<td></td>
</tr>
<tr>
<td>AS Java</td>
<td>5,700</td>
<td>5,700</td>
<td>11,400</td>
<td>9,400</td>
<td>5,000</td>
<td>5,000</td>
<td></td>
</tr>
<tr>
<td>MR Unix</td>
<td>5,000</td>
<td>4,300</td>
<td>8,000</td>
<td>7,000</td>
<td>5,100</td>
<td>4,300</td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>27,300</td>
<td>20,500</td>
<td>31,000</td>
<td>24,200</td>
<td>17,300</td>
<td>17,300</td>
<td></td>
</tr>
<tr>
<td>HD</td>
<td>5,700</td>
<td>4,800</td>
<td>10,400</td>
<td>8,000</td>
<td>5,000</td>
<td>4,800</td>
<td></td>
</tr>
<tr>
<td>Voice</td>
<td>4,300</td>
<td>4,500</td>
<td>7,900</td>
<td>5,774</td>
<td>3,861</td>
<td>3,100</td>
<td></td>
</tr>
<tr>
<td>WAD</td>
<td>3,100</td>
<td>3,100</td>
<td>7,900</td>
<td>5,774</td>
<td>3,861</td>
<td>3,100</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>108,150</td>
<td>100,500</td>
<td>131,850</td>
<td>102,374</td>
<td>68,361</td>
<td>71,700</td>
<td>28,800</td>
</tr>
</tbody>
</table>

Example: Overview of all activities
Step 7: Define the metric per activity for performance optimization

<table>
<thead>
<tr>
<th>Metric</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD cobol</td>
<td>200</td>
<td>194</td>
<td>170</td>
<td>125</td>
<td>90</td>
</tr>
<tr>
<td>AS cobol</td>
<td>21</td>
<td>17</td>
<td>30</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>Mainframe</td>
<td>12,500</td>
<td>9,600</td>
<td>8,000</td>
<td>6,800</td>
<td>5,500</td>
</tr>
<tr>
<td>AD Java</td>
<td>167</td>
<td>156</td>
<td>325</td>
<td>236</td>
<td>146</td>
</tr>
<tr>
<td>AS Java</td>
<td>33</td>
<td>29</td>
<td>53</td>
<td>33</td>
<td>14</td>
</tr>
<tr>
<td>MF Unix</td>
<td>88,000</td>
<td>54,400</td>
<td>52,000</td>
<td>64,000</td>
<td>47,000</td>
</tr>
<tr>
<td>DC</td>
<td>2,600</td>
<td>2,450</td>
<td>1,500</td>
<td>1,200</td>
<td>900</td>
</tr>
<tr>
<td>HD</td>
<td>39</td>
<td>61</td>
<td>15</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>Voice</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WAD</td>
<td>716</td>
<td>611</td>
<td>418</td>
<td>296</td>
<td>219</td>
</tr>
</tbody>
</table>

Example: Overview of all metrics

Step 8: Compare the performance with all industries’ IT metrics (optional)

<table>
<thead>
<tr>
<th>Respondent Profile</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of respondents</td>
<td>-</td>
<td>416</td>
<td>-</td>
</tr>
<tr>
<td>Average revenue</td>
<td>$1,340,000</td>
<td>$1,383,000</td>
<td>$1,480,000</td>
</tr>
<tr>
<td>Average number of employees</td>
<td>-</td>
<td>5,120</td>
<td>-</td>
</tr>
<tr>
<td>Average number of IT professionals in the workforce</td>
<td>-</td>
<td>261</td>
<td>-</td>
</tr>
</tbody>
</table>

Key Metrics
- IT operating budget as a percentage of revenue: 3.10% 2.78% 2.93%
- IT capital budget as a percentage of revenue: 0.99% 1.11% 1.02%
- Estimate of percentage of total spending on part of central IT budget: 28%
- Average IT operating budget per employee: $6,953 $6,604
- Average percentage of IT employees: 5.12%
- Percentage of internal IT employees: 82%

Example: Industry high-level metrics
World class performing better on the full mix of the core value drivers (EQFIV)

- World class IT organizations increase the competitiveness of a company, is performing better than other companies on the full mix of the core value drivers (EQFIV):
  - Efficiency (produce for low cost/unit)
  - Quality and effectiveness (do it the first time right)
  - Flexibility (ensuring availability of an attractive product range = small lot high frequency)
  - Innovation (having the latest developments add to the value of the product)
  - A chain of values or Network value
- (ITIL) Process Performance benchmark
- Service Level Agreement (SLA) benchmark
- Prize benchmark

Questions?
Monitoring and Debugging of Web applications
Quality in Practice - Tools and methodologies used at a software company

Martijn van Berkum

GX, Nijmegen, The Netherlands

Abstract

The concept quality, and in particular, quality of software, has a coherent theory. To use this theory in practice, however, is a lot harder. To reach a defined quality level, good methodologies and tools are a necessity. In this presentation we will give an overview of the methodologies and tools used at GX. GX is a software company that delivers technology for the management of high-traffic dynamic websites. We will present and sometimes shortly demonstrate several tools used at GX to log and monitor live websites, do performance analysis and debug, check, test, validate and manage our code.

We will give an evaluation of methodologies and tools that we use in practice and worked for us, and those that didn't work.
Discount Usability Testing

Rob Hendriks

rhe@improveqs.nl

Why usability?

- Reduce number of user errors
- Increase efficiency
  - 83% of website visitors leave if many clicks have to be made
- Increase user satisfaction
- Increase ease of learning
- Increase trust
- Decrease support costs
  - Company spent $900,000 due to difficulties installing printer driver
- ……
The usability gap

- Usability often a critical success factor
  - 25% of IT-projects fail
  - 42% of the code is user-interface

- Usability testing should be compulsory, but...

Usability in action

- ATM “...only usable on an overcast day...”
- Video recorder
- Remote control (two user groups)
- Navigation system can’t be used wearing sunglasses

Focus

Broad view OR narrow focus

How do I leave the help?

“Can users carry out their tasks?”
Testing usability

- A few possible ways of testing usability:
  - Heuristic evaluation
  - Questionnaire based (SUMI)
  - Simulation
  - User acceptance testing
  - Usability lab

When to test usability?

1. Define and validate usability requirements
2. Review the specification and the design usability
3. Verify the implementation

Heuristic evaluation

Simulation

SUMI

UAT

(SUMI) Vermeeren

Time spent on development

Room for improvements

Costs of improvements

(A.P.O.S. Vermeeren)
Heuristic evaluation

- Testing / evaluating the usability of a software product against a number of usability principles
  - E.g. 10 usability heuristics (Jakob Nielsen, 1993/1994)
- Identifies usability problems early in the design process in a quick manner
- Predicts real life usability problems !!
- Often on prototypes
- Both broad and narrow focus

10 Heuristics

1. Visibility of system status
2. Match between system and real world
3. User control and freedom
4. Consistency and standards
5. Design useful error messages
6. Recognition rather than recall
7. Flexibility and efficiency in use
8. Aesthetic and minimalist design
9. Error prevention
10. Help and documentation
Case Governmental department

- Financial application
- GUI guideline defined
- Heuristic evaluation executed on prototype
  - 20 screens
  - 114 findings
- SUMI executed as zero measurement

SUMI

- Software Usability Measurement Inventory
- Broad focus: user satisfaction
- Well founded questionnaire: based on practical research (MUSiC)
- Referred to in ISO 9126 and ISO 9241
- The user scores are standardised by using a reference database
- Quantitative, objective information of users’ subjective attitude to six usability aspects
Case Schiphol Airport

- Replacement CISS
  - Central Database System / Information Broker

- Actions carried out
  - Define GUI guidelines based on Heuristics
  - Prototypes and final GUI tested by means of Heuristic evaluation
  - SUMI test executed on current and new system
  - Acceptance criteria:
    New global SUMI score no more than 10% less than current global SUMI score
Result Schiphol Airport

Little effort – great benefit

Conclusion

- Usability is often stated as important
- It has a high contribution to the product’s success
- Usability is however hardly tested in practice
- Relatively cheap techniques are available with a high benefit
- Start as early as possible

Thank you
Testing at Nucletron

The role of integration testing in an international organisation

Nucletron

- **Description**
  - Radio Therapy treatment for Oncology
  - Part of Delft Instruments Medical
  - Offering Hardware, Software and Solutions to Oncology departments all over the world
  - About 480 employees in 16 countries
  - Unique knowledge
Research and Development

- **Treatment planning software**
  - Development of special algorithms for
    - Radiotherapy doctor
    - Physicists
  - Treatment plan analysis software
- **Special requirements**
  - Highly innovative
  - Mixture of medical physicists/informatics
  - Co-operation with
    - Other small development companies
    - Research groups in universities

Specific situation

- **Integration**
  - Software portfolio is coming from take-overs

- **Chances at client sites**
  - Decision is slowly moving from Radio Therapy department to ICT department (application infrastructure is being standardised)

- **Solution**
  - Organisation directed to
    - Communication
    - Integration
Organisational design

- Build in conflict handling
- Making testing a central element in design
- Making integration testing a central responsibility

Integration test

- Central responsibility for integration test
Organisation

- Building of test bed at the moment of design
- Developers are using test cases of test environment when possible
- Development center has separate test co-ordinator for application testing based on test environment (reporting in development center)
- Development center accepts software from development group (internal or external)
- Head office uses total test bed based on workflows in hospital RT centers to test total system

Testing function

- Global integration testing, acting as
  - Quality check
  - Backwards pressure on the development organisation
  - Conflict between product management and level of quality
Methods

- First step was the improvement of the development practices: CMM Level 2 (based on RUP best practices)

- Second step is the improvement of the test practices: (TMM Level 2). For the multi site environment:
  - Application testing at development sites;
  - Integration testing at HQ.

- Emphasis on quality (health care software)
PHILIPS

Privacy and Security in Healthcare

Milan Petković
Information and System Security Department
PHILIPS Research Laboratories

Outline

- Introduction
  - Healthcare and IT
  - Electronic health records
- Healthcare Privacy and Security Issues
- Privacy and Security Requirements in Healthcare
- State-of-the-art Technologies
- Novel approaches
  - Operations on Encrypted Data
  - Role-based Access Control
Healthcare and IT

- Healthcare – an important service sector (the largest service sector in the US economy)
- The healthcare industry is under constant pressure to become more efficient
- Driving factors of further IT deployment in Healthcare
  - An increasing number of elderly people and increasing costs for healthcare
  - Changes in healthcare (chronic diseases make most of the costs, people die because of chronic diseases and degenerative illness): reactive -> proactive
  - The tendency for the enhancement of the quality of healthcare are driving deployment of IT in Healthcare
- Most important new applications of IT: electronic health records, telemmedicine, clinical decision support systems including pervasive computing solutions, etc.

Mistakes in the delivery of Healthcare

- 44,000 to 98,000 people die in USA hospitals each year as result of medical errors that could have been prevented - To err is human, IOM 1999
- 6.1% (800,000) people in the Netherlands had wrong medical treatment - NICTIZ (Nationaal ICT Instituut in de Zorg) NIPO Rapport, 2004, “Fouten worden duur betald”
  - Reasons: (1) no insight in the medical record of the patient (2) wrong maintenance of the medical record of the patient
- Nature of mistakes:
  - Wrong medication (44%)
  - No treatment because of the lack of information (25%)
  - Wrong surgery or treatment (24%)
  - Planning mistake (18%)
- Effects:
  - Emotional problems - 400,000 people
  - Physical problems – 250,000 (125,000 permanent)
  - Partial (14,000) or complete (36,000) invalidity
- Financial effect – euro 1.4 billion a year
National EHR Systems

Recently a number of countries have introduced plans for national electronic record (EHR, EPR, EMR, EPD, etc) systems

- UK: 2005 National Health Service all medical records to be available electronically to all UK citizens
- Finland: fully functional in 2007
- Australia: 2000 HealthConnect first step in providing a national EHR system
- Germany: 1995 DM500M smartcard that hold specific health information for the individual. Also smartcard technology for stakeholders that amongst other feature contains digital signatures
- France: SESAM-Vitale and pilot project at Montreuil sur Mer
- Belgium: 1999 smartcard that encodes fingerprints to all citizens
- US: in preparation, currently trials in some areas. HIPAA regulates the transmission of medical information in an electronic format

Electronic Health Records and Security

- Healthcare information security – one of the key obstacles to the EHR concept
- EHR goes online, but internet is the source for about 70% of all hacking attempts.
- Legislation around security and privacy (HIPAA to 10 years in prison for selling the data)
- Therefore, ensuring adequate information security is one of the main IT priorities in Healthcare.
HIPAA

- **Health Insurance Portability and Accountability Act**

- HIPAA is about
  - Privacy
  - Security
  - Electronic transactions & code sets (e.g. diagnosis and procedure)

- **Covered entities:**
  - Health Plans
  - Health Care Providers
  - Health Care Data Clearinghouses

- Covered entities must comply with HIPAA standards for privacy, security, code sets, electronic transactions, etc.

HIPAA Requirements - Privacy

- **Covered entities**
  - cannot disclose or use PHI (Protected Health Information) for specific purposes without the individual’s consent
  - must have policies to minimize PHI disclosure (except when disclosure is needed for treatment purposes)

- They must comply with patient right to:
  - Ask for a copy of health records
  - Have corrections added to the record
  - Receive a report on when and why his health information was shared
  - Decide if he wants to give his permission before his health data can be used for certain purposes
  - Ask that his health information not be shared
  - …

- Compliance with Privacy standards is required by April 14, 2003
HIPAA Requirements - Security

- Meant to protect PHI from improper access, alterations and loss
- Covered Entity must implement administrative, physical and technical security standards to
  - Ensure the confidentiality, integrity, and availability of all electronic protected health information
  - Protect against any reasonably anticipated threats or hazards to the security or integrity of such information
  - Protect against any reasonably anticipated uses or disclosures of such information
- Required and addressable standards implementation
- Compliance with Security standards is required by April 21, 2005

Technical safeguards standards (HIPAA)

- Access Control
  - Unique user identification (R) – user, role-and/or context-based
  - Emergency aspect procedure (R)
  - Automatic logoff (A)
  - Encryption and decryption (A)
- Audit Control
  - Record activities regarding PHI
- Integrity
  - Implement e-systems that verify that PHI is not altered or destroyed
- Person or entity authentication
- Transmission Security
  - Assess risks
  - Implement integrity control and encryption (A)
Privacy and Security Requirements

- **Data Integrity**
  - Basic requirement: integrity of healthcare data must be protected when data is stored, transmitted and operated upon.

- **Data Confidentiality**
  - Basic requirement: healthcare data should be protected against improper disclosure when stored, transmitted and operated upon.
  - More complex (e.g., fine-grained) access control models should be supported.
  - Long-term protection of health data.

- **Data Availability**
  - Basic requirement: healthcare data must be available to authorised parties, whenever and wherever needed.

- **Trade-off between data confidentiality and availability**
  - Basic requirement: healthcare data should be available to authorised parties only if needed.
  - There may be the possibility to perform operations as well as queries on encrypted data when such operations and searches are performed by untrusted parties.
  - Easy and quick authentication of healthcare providers and identification of patients.

- **User awareness and control on data use**
  - Basic requirement: patients’ rights regarding his health information should be protected by the system.
  - Auditing.

- **Accountability and Non-repudiation**
  - Basic requirement: accountability and non-repudiation should be supported in the system concerning all entities.
  - The system should provide accountability by means of controls, processes and policies that allow the system to trace actions to their source.

Privacy and Security Technologies

- **Data Integrity**
  - Digital signatures.
  - Forward-secure digital signatures.
  - Methods to prevent deletion.
  - Backups.
  - Software Integrity and Trusted Computing.

- **Data Confidentiality**
  - Encryption.
  - Operations on encrypted data.
  - Zero-Knowledge.
  - Secret sharing.
  - Access Control.
  - Authentication.
  - Anonymization.
  - Privacy preserving data mining.
  - Private information retrieval.

- **Data Availability**
  - Electronic health data management.
  - Redundancy.

- **Trade-off between data confidentiality and availability**
  - Easy and convenient authentication.
  - Secret sharing, k out of n.
  - Databases and encryption.

- **User awareness and control on data use**
  - Auditing mechanisms.
  - Digital Rights Management.
Privacy protection in personalized services
Stefan Maubach, Milan Petkovic, Verus Pronk, Pim Tuyls, Wim Verhaegh

• Health monitor example

```
user
profile
bph 120
bpl 70
hr 60
age 27
wt 70

server

match

reference profile 1

bph 120
bpl 70
hr 60
age 27
wt 70

similarities

healthy

similarities

healthy

low salt

diet

similarities

healthy

low salt

diet
```

Encrypted health monitor

• User data encrypted before going out
• Only user can decrypt end result

```
user
profile
bph 120
bpl 70
hr 60
age 27
wt 70

server

encrypted profile

encrypted profile

match

encrypted profile

encrypted profile

encrypted match

encrypted match

similarities

healthy

similarities

healthy

low salt

diet

similarities

healthy

low salt

diet
```
Technical challenge

- Conflict: encrypt user data ↔ do personalization
- Use encryption scheme with special properties:
  \[ \text{message } x \rightarrow \text{enc}(x) \cong b^x \]
  \[ \text{enc}(x) \cdot \text{enc}(y) \cong b^x \cdot b^y \cong b^{x+y} \cong \text{enc}(x+y) \]
  \[ \text{enc}(x)^y \cong (b^x)^y \cong b^{x\cdot y} \cong \text{enc}(x\cdot y) \]

- Enables all kinds of computations on encrypted data

Personalized RBAC

M. Petkovic, C. Conrado, M. Hammoutene

Alice and Bob are GPs
GP has access to part x of EHR data of any patient
Exceptions: "Alice has access, not Bob"
Personalized RBAC

Has this GP extra rights?

Ex: GP17 has no access to blocks 18 and 19

Exception List List:

GP(17,18,19)

Role

Blocks

ID of the concerned GPs

He has has not default rights

Cryptographically Enforced Personalized RBAC:

Efficient Key Management

Alice, Bob, Carol, and David can decrypt the data
Cryptographically Enforced Personalized RBAC: Efficient Key Management

\[ [\text{Data}]_k : [K]_A [K]_B [K]_D \]

Alice, Bob and David can decrypt the data, not Carol.

Cryptographically Enforced Personalized RBAC: Efficient Key Management

\[ [\text{Data}]_k : [K]_A [K]_B [K]_C \]

Alice, Bob and David can decrypt the data, not Carol.
Conclusions

➢ Privacy and Security in Healthcare: an important problem that needs research because of:
  ➢ Healthcare information is typically very privacy sensitive
  ➢ Specific privacy and security requirements from legislation
  ➢ Specific characteristics of healthcare data (e.g. long term value)

➢ Some specific solutions
  ➢ Private profile matching
  ➢ Crypto-enforced Personalized RBAC
Improving the Quality of Protocol Standards

Judi Romijn
OAS group, TU/e
jromijn@win.tue.nl

The NWO-funded project 'Improving the Quality of Protocol Standards' aims at protocol descriptions in standards which are formal yet readable, and formally correct. What we stated in our project proposal in 2001 has been confirmed by our participation in three protocol standards: the quality of protocol descriptions in standards is poor, and our contribution is dearly needed. In this project we have indeed improved the quality of the standards involved, and have found inspiration for theoretical research based on the methods used in the standardisation process.

We have worked on three (families of) protocol standards.

- **IEEE 1394.1 FireWire Bridges**
  This standard defines how IEEE 1394 serial buses are linked with bridges. To manage the larger network of buses, the bridges engage in a distributed spanning tree protocol called net update. By formalising and analysing net update, we uncovered many mistakes, unclarities and omissions, and even one crucial bug (non-termination of the protocol) in the draft standard description. We have applied model checking to parts of the protocol with the tools Spin, muCRL and CADP, and we have formally constructed an abstract version of the protocol and a variant with the Feijen/vanGasteren, Owicki/Gries and Dijkstra methods. All formal construction proofs have been checked in the theorem prover PVS.
  The Spin model checking work has led to new theory about guiding simulation into the direction of suspected errors, which is directly applicable to Spin experiments. The theory has been proved correct and besides simulation, also allows for verification experiments on guided models, such that errors found in the guided model are also errors of the original model.
  The resulting IEEE standard contains about twice as much text describing the net update protocol. We have participated in the Ballot Response Committee (BRC) which adjusted the draft standard after the first ballot. Based on our feedback, the resulting description is of much higher quality, and contains a new subprotocol that deals with the errors we found. By our suggestion, the standard includes an appendix with correctness properties (intended functionality) for the net update protocol, enabling manufacturers to check whether their implementation of 1394.1 works correctly.

- **ISO/IEEE 1073 Medical Device Communication**
  In this standard concerns communication between medical devices. We have participated in the working groups that contribute to three of its protocol standards. Although medical systems must be extremely reliable under all circumstances, before our involvement no formal analysis was performed during the development of this family of standards.
  Some protocols were defined through state tables and textual descriptions. Our formal analysis with Spin revealed various discrepancies and undesired behaviors. The extended and corrected state tables have been incorporated in the standard.
  One standard initially contained a set of scenarios in the shape of Message Sequence Charts (MSCs) and textual descriptions. So there were no state tables, and only the basic scenarios were given (in terms of the MSCs). It turned out that with current MSC techniques, one cannot properly extract state tables from these MSCs; this is caused by a phenomenon called non-local choice. Based on this case study, we have initiated a new research direction by proposing several ways to implement MSCs that contain non-local choice. By applying this, we have extracted state tables from the MSCs in this standard, and these state tables will be incorporated in the standard.
  We have also defined a new semantics for MSCs based on partial orders, which allows deadlocks and shows the completeness or our earlier classification of choice-related problems in MSCs.

- **ANSI HL 7 Medical Device Communication**
  Health Level Seven (HL7) is an ANSI standard that provides a framework for electronic health information. Our work has focused on the HL7 specification of a communication protocol that enables health-care applications to exchange key sets of data. We have created state diagrams for this protocol, by combining message sequence chart (MSC) descriptions of a number of intended behaviors in the current draft standard. We have reused and extended our MSC theory from the 1073 case study in order to solve arising problems such as deferred behaviour and non-local choice. Our work has revealed a number of inconsistencies in the view and intention of the developers. This has initiated much discussion in the working group, which is yet to converge to a completely new proposal.

**Project information**

People: Romijn (project leader), Goga, Mooij, Wesselink
Constructing Correct Parallel Programs

The Feijen/van Gasteren method is an assertion-based method for constructing parallel programs that are correct by design. Typically, it involves:

1. **Formal specification of the program**: A high-level specification is created, describing the desired functionality.
2. **Construction of the program**: The program is constructed from the specification, using an algorithm that is derived from the specification.
3. **Verification of the program**: The correctness of the constructed program is verified using automated theorem provers.

Selected Methods and Tools

- **Feijen/van Gasteren method**: A widely used method for constructing parallel programs that are correct by design.
- **Owicki/Gries proof obligations**: Used to verify the correctness of the program.
- **Automated theorem provers**: Tools like POCL, PVS, and Isabelle are used to automate the verification process.
- **Guiding Spin simulation & verification**: A technique for limiting the explored state space by guiding the simulation towards specific parts of the state space that are more likely to contain errors.

Case Studies

- **The IEEE 1394.1 case study: Firewire Bridges**: A case study focusing on the development of protocols for Firewire bridges, where the correctness of the protocols is crucial.
- **The ISO/IEEE 1073 case study: Medical Device Communications**: A case study focusing on the development of protocols for medical devices, where the reliability and safety of the protocols are critical.

Formal specification and analysis of protocols (distributed algorithms) during the standardisation process, which leads to:

- Better descriptions in standards
- Inspiration for improving formal methods and tools

Some protocols in IEEE 1073 are described by scenarios, like the network model. However, for constructing correct parallel programs, the choice construct poses problems, e.g., in case of non-local choice. The typical result is an implementation deadlock!

Other formal methods are used to prove the correctness of the program, such as model checking and automated theorem provers. By checking properties against specifications, the correctness of the program can be verified.

The Feijen/van Gasteren method Tool Support

- **Proof generator**: Generates proof obligations from the specification.
- **Automated theorem prover**: Verifies the correctness of the program.
- **Guiding Spin simulation & verification**: Helps limit the explored state space.

Guiding Spin simulation & verification

To verify distributed systems, an effective technique involves the development of state space exploration tools. This method helps in identifying potential errors in the system by simulating the system over its state space and checking if any errors are present.

The IEEE 1394.1 case study: Firewire Bridges

- **Construction and verification**: Constructed algorithms and verification of correctness through graphs.
- **Automated theorem provers**: Automated theorem provers like POCL, PVS, and Isabelle are used to prove the correctness of the constructed algorithms.
- **Guided Spin simulation & verification**: Using guided Spin simulation, we can explore only a part of the state space that is more likely to contain errors.

The ISO/IEEE 1073 case study: Medical Device Communications

- **Construction and verification**: Construction of correct parallel programs and verification of correctness through graphs.
- **Automated theorem provers**: Automated theorem provers are used to prove the correctness of the constructed algorithms.
- **Guided Spin simulation & verification**: Using guided Spin simulation, we can explore only a part of the state space that is more likely to contain errors.

The BG/IEEE 1073 case study: Medical Device Communications

- **Construction and verification**: Construction of correct parallel programs and verification of correctness through graphs.
- **Automated theorem provers**: Automated theorem provers are used to prove the correctness of the constructed algorithms.
- **Guided Spin simulation & verification**: Using guided Spin simulation, we can explore only a part of the state space that is more likely to contain errors.

The BG/IEEE 1073 case study: Medical Device Communications

- **Construction and verification**: Construction of correct parallel programs and verification of correctness through graphs.
- **Automated theorem provers**: Automated theorem provers are used to prove the correctness of the constructed algorithms.
- **Guided Spin simulation & verification**: Using guided Spin simulation, we can explore only a part of the state space that is more likely to contain errors.

The IEEE 1394.1 case study: Firewire Bridges

- **Construction and verification**: Construction of correct parallel programs and verification of correctness through graphs.
- **Automated theorem provers**: Automated theorem provers are used to prove the correctness of the constructed algorithms.
- **Guided Spin simulation & verification**: Using guided Spin simulation, we can explore only a part of the state space that is more likely to contain errors.

The ISO/IEEE 1073 case study: Medical Device Communications

- **Construction and verification**: Construction of correct parallel programs and verification of correctness through graphs.
- **Automated theorem provers**: Automated theorem provers are used to prove the correctness of the constructed algorithms.
- **Guided Spin simulation & verification**: Using guided Spin simulation, we can explore only a part of the state space that is more likely to contain errors.
Monitoring the Quality of UML Architecture and Design Models using MetricView

Within the EmpAnADa project (www.win.tue.nl/empanada)

The primary goal of this project is to develop methods to improve practitioners’ use of the UML and model quality. A known problem of UML is the lack of formality in usage of the language.

Survey results show that UML is used rather loosely and incompleteness of models causes problems such as miscommunication. Despite of the fact that there are no techniques to assess model completeness, it is the most frequently reported criteria to finish the modeling phase. We have developed a rule-set to assess model completeness. This rule-set was applied to industrial UML models. The results of these case studies show that lacking completeness of UML models is a critical issue in practice. The rule-set can assess model completeness and inconsistency.

MetricView visualizes the results of checks and metrics analysis on top of existing UML models.

Relations between Measurement in UML Models and Source Code

Controlling Completeness and Consistency of UML Models

- Completeness is the no. 1 criterion to stop modeling, nevertheless no method exists to quantify it.
- Methods to evaluate model completeness and find incomplete and inconsistent spots were developed and implemented in a tool.
- Large scale industrial case studies were conducted:
  - The degree of incompleteness in real world models is very large.
  - Example: a model showed 58,7% of all messages in Sequence Diagrams did not correspond to methods.

Measuring Quality Attributes in UML Models

- Metrics were developed that combine information from different diagram types. Examples:
  - Complexity of a class based on state transitions and method definitions
  - Number of use cases per class to measure functional cohesion
  - Number of classes per use case supports in prioritizing use cases

MetricView:

- Combines Model & Metrics Visualization
- Exposes Design Smells
- Custom Quality Models

Monitoring the Quality of UML Architecture and Design Models using MetricView

Within the EmpAnADa project (www.win.tue.nl/empanada)

The primary goal of this project is to develop methods to improve practitioners’ use of the UML and model quality. A known problem of UML is the lack of formality in usage of the language.

Survey results show that UML is used rather loosely and incompleteness of models causes problems such as miscommunication. Despite of the fact that there are no techniques to assess model completeness, it is the most frequently reported criteria to finish the modeling phase. We have developed a rule-set to assess model completeness. This rule-set was applied to industrial UML models. The results of these case studies show that lacking completeness of UML models is a critical issue in practice. The rule-set can assess model completeness and inconsistency.

MetricView visualizes the results of checks and metrics analysis on top of existing UML models.

Relations between Measurement in UML Models and Source Code

Controlling Completeness and Consistency of UML Models

- Completeness is the no. 1 criterion to stop modeling, nevertheless no method exists to quantify it.
- Methods to evaluate model completeness and find incomplete and inconsistent spots were developed and implemented in a tool.
- Large scale industrial case studies were conducted:
  - The degree of incompleteness in real world models is very large.
  - Example: a model showed 58,7% of all messages in Sequence Diagrams did not correspond to methods.
Information Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands
http://is.tm.tue.nl/

ProcessMining
processmining.org

The ProM suite

/department of technology management
SpecTEC: Specification Tooling for Embedded software Components

TU/e participants: dr. R.Kuiper, prof. dr. J.C.M. Baeten, dr. E.J.Luit
EES: 5141 Progress: ir.L.C.M. van Gool (aio), dr. S.(E.E.) Roubtsov(a) (postdocs)
Industrial partner: Philips Natlab dr. H.B.M. Jonkers (user also: Océ )

Aim:
Tool support for consistency of ISpec Interface Specifications [1].

Motivation:
The ISpec Interface Specification approach is developed and used at Philips for the design of complex embedded systems. It entails many views and diagrams (related to Rational’s Unified Modelling Language) and notions of refinement and composition (related to Object-Oriented Development).

ISpec provides one model to keep all these different descriptions consistent. This model is described using structured pre / post / action clause / invariant templates. ISpec templates are user-friendly in that they allow plug-in use of different specification languages, at different levels of formality, ranging from natural languages to logics.

Innovation by SpecTEC (see picture):
- Formal underlying semantic model
- Tooling to support construction and consistency of ispecs

Approach:
- Formal underlying semantic model
  - relational calculus denotational model
  - method call/return representation
  - inheritance/composition formalisation
  - proof system
  - connection to Hoare style semantics
- Tooling
  - Visio based tool
    - XML representation of model
    - Interface-Role Diagrams + ISpec checks
    - Regular Expressions for Action Clauses
    - Sequence Diagram checks (current work)
- Development of the Visio-based tool will be continued
- Successful Océ pilot
- Tool to be used at Philips

http://www.win.tue.nl/calisto/ (under construction)
Yet Another Smart Process Editor

- design goal: make simple things simple (editing + simulation)
- no attempt to replace existing tools
- no attempt to offer all potentially useful features

- specify: draw process flows as diagrams
- verify: execute (simulate) manually / automatically
- feed them to other tools (e.g. for analysis)
VALENS EE is a tool that offers functionality to assess the quality of a set of (logical) rules. The rules can be verified to check their logical correctness, visualized to increase understanding of the rules and validated to see if the behavior of the rules conforms to the desired behavior of the rules.

The user interacts with VALENS to verify and validate the rules. After verifying and validating the rules the user has an understanding of the quality of the rules.

VALENS is complementary to any rule-editing environment. The rules can be exported to LibRT’s rule base markup language (RBML) and viewed in VALENS. RBML is freely available from the LibRT website (www.librt.com). For assistance with transformations from a rule language to RBML, contact us at info@LibRT.com.

Verification is the process that aims for the detection of inconsistency, incompleteness or redundancy in a set of business rules without consideration of the ‘meaning’ of the rules. This process does not take into account the correctness of the business rules, i.e. whether their effect is indeed the intention of the business. If verification can prove that the rules are logically consistent and complete, the rules may still lead to incorrect results (but they will do so in a consistent way).

Validation is the process that aims for the detection of incorrect results or undesired behavior. The most common way of validating rules is to just pass the (changed) rules to another member of the organization. VALENS supports this process by rule visualization.

In the context of an IT-project validation is often done by testing the application and assessing the results, or comparing the results with previous results that were believed or known to be correct. VALENS supports this process with functionality to test rules. Validated rules or business cases are approved by members of the organization responsible for the rules.

Why is verification & validation important?
You know your rules, but do you know if they are right? Humans have a hard time understanding hundreds, thousands or even twenty rules with complex interactions. LibRT’s VALENS can help you in assessing the quality of rules by delivering detailed information on the completeness and consistency of your rules and the consequences of rule changes to existing rules and predefined cases. Taking new rules into production can now be a rational decision based on the information provided by VALENS.

Valens primary aims at the business experts
Establishing the quality of rules is important for every organization that needs to communicate and process complex regulations, expertise and guidelines that can be translated to rules. Business users who define rules can use VALENS to assist them in the delivery of consistent, complete and correct rules. VALENS can also be used by IT departments, who want to establish the quality of rules prior to implementation, decreasing development and test time.

About LibRT
LibRT supports enterprise clients and software vendors with products and services targeted at effective knowledge management in business applications. Based in the Netherlands, LibRT does business throughout Europe and North America with a network of partners providing complementary technologies, services, and delivery channels. Among the company’s innovative products and designs is LibRT VALENS, the industry’s first independent product targeted at verifying and validating business rules created in third-party business rules management systems.

LibRT focuses on supporting the delivery of high quality business rules. The company is a leading proponent and driver of rule qualification standards, with active participation in ISO certification requirements for quality assessment of rules and proper generation and selection of business rule test cases. Information about products, services, clients, and partners can be found at www.librt.com.

*The Dutch Tax and Custom Administration has recognized the need for an efficient formalization of legislation in declarative models. We verify our legal knowledge representations to ensure proper law enforcement. LibRT has been one of the partners that assists us with our approach in insuring legal quality with their verification engine VALENS.*

Prof. Dr. T. van Engers Program manager Belastingdienst
screenshot
This screenshot shows the list of attention points detected during verification and two different visualizations of the same rules.

feature list
This table shows the features supported by VALENS enterprise edition.

<table>
<thead>
<tr>
<th>verification</th>
<th>visualization</th>
<th>validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambiguity (conflicts)</td>
<td>rules</td>
<td>enter test case</td>
</tr>
<tr>
<td>self contradiction</td>
<td>decision tree</td>
<td>generate test cases (planned for version 3.2)</td>
</tr>
<tr>
<td>circular reasoning</td>
<td>decision table</td>
<td>save test case</td>
</tr>
<tr>
<td>subsumption</td>
<td>fishbone diagram</td>
<td>after rule change:</td>
</tr>
<tr>
<td>incomplete range checking</td>
<td>dependency graph</td>
<td>- assess correctness of test case</td>
</tr>
<tr>
<td>incomplete value checking</td>
<td>scenario diagram</td>
<td>- assess completeness of test case</td>
</tr>
</tbody>
</table>

options
- double click on attention point to view details
- view details in visualization view

- compress view
- navigate to definitions of variables
- substitute definitions of variables
- support for long names
- zoom-in and zoom-out
- collapse or expand branches of tree

- see the list of input variables
- choose from list of input values
- inspect list of intermediate values

contact
For more information, information on pricing, resellers or demo’s use the following contact information or go to www.librt.com.

silvie spreeuwenberg  T +31 (0)20 422 28 93 (GTS +1.00) silvie@librt.com
Mercury Business Process Testing™ is a complete system for test automation, enabling non-technical subject matter experts to become an integral part of the quality optimization process.

Do you find that most of today’s functional testing products are too dependent on the programming to enable broad adoption in your team? Do miscommunications and different priorities between subject matter experts and quality engineers result in time-consuming test rework? Have you found that limited subject matter expert involvement during testing leads to defects and breakdowns in critical business processes? Are defects found in production instead of by your functional testing team — hurting your group’s credibility?

Mercury Business Process Testing is the first complete role-based test automation system to overcome these challenges and bridge the quality chasm between subject matter experts and quality engineers. Business Process Testing is the first Web-based test automation solution designed from the ground up to enable subject matter experts to build, data-drive, and execute test automation without any programming knowledge.

Our solution reduces the overhead for automated test maintenance and combines test automation and documentation into a single effort. You are able to measure the quality of application deliverables from abstract business definitions defined within the Business Process Testing framework.

In our role-based solution, subject matter experts focus on creating high-level test flows that mirror actual business process, while quality engineers concentrate their efforts on areas than enable automation.

How it Works

Business Process Testing improves on technology known as “Table-Driven” or “Keyword Driven” testing. This next-generation approach to test automation introduces best practices into test design, and enables a complete solution for test design, maintenance, and execution. The system introduces the concept of reusable business components that drastically reduce test maintenance and improve test creation efficiency.
The Business Process Testing system is role-based, allowing non-technical subject matter experts to define test cases without the need for programming or scripting. Subject matter experts define test flows through a Web-based interface by declaring what steps to take and what data to use. By deploying a test-framework approach to test automation, QA engineers are focused on enabling automated testing assets.

Our system allows you to begin quality assurance efforts earlier in the lifecycle of application development. A major benefit is that it simplifies the creation of tests by leveraging a new technology, known as “Keyword Driven Testing,” which allows English representation of test cases. This technology eliminates the need for scripting programming when building test assets.

Through the business component technology, Business Process Testing also streamlines the maintenance of testing assets, as both manual and automated testing definitions can use highly reusable business definitions. These business components centralize test maintenance in one repository. Furthermore, the system generates test-plan documents (in Word format) based on test definitions developed using Business Process Testing.

Business Process Testing automatically generates Test Plan documents in industry-standard Microsoft Word format.

Our system allows you to begin quality assurance efforts earlier in the lifecycle of application development. A major benefit is that it simplifies the creation of tests by leveraging a new technology, known as “Keyword Driven Testing,” which allows English representation of test cases. This technology eliminates the need for scripting programming when building test assets.

Through the business component technology, Business Process Testing also streamlines the maintenance of testing assets, as both manual and automated testing definitions can use highly reusable business definitions. These business components centralize test maintenance in one repository. Furthermore, the system generates test-plan documents (in Word format) based on test definitions developed using Business Process Testing.

Business Process Testing sits on top of a Web-enabled enterprise-class technology platform that is fully integrated into Mercury Quality Center™. Our solution combines ease of use, scalability, fast deployment, and rich functionality to support the entire development lifecycle.

With Business Process Testing, you can test more thoroughly and, in less time, catch more defects and release better applications than previously possible.

Part of Mercury Quality Center

Mercury Business Process Testing is part of Mercury Quality Center™, an integrated set of software, services, and best practices for automating key quality activities, including requirements management, test management, defect management, functional testing, and business-process testing.

FEATURES AND BENEFITS

• Allows non-technical subject-matter experts to quickly build, data drive, and document tests in one Web-based system.
• Eliminates the need for programming to define business process flows due to script-free test design.
• Reduces the effort required for test maintenance by deploying centralized Business Components.
• Facilitates organizations to start test automation earlier in the development lifecycle, even before an application is delivered to Quality Assurance.
• Automatically generates Test Plan Documentation through an innovative Auto-Documentation mechanism.
• Enables QA efforts to best leverage talent through specific roles and responsibilities.
• Enables User Acceptance Test (UAT) to deploy automation with minimal training.
• Centralizes test-maintenance so application changes are automatically propagated through automated test assets.

Business Process Testing automatically generates Test Plan documents in industry-standard Microsoft Word format.
Today’s Challenges Make Enterprise Applications Prone to Debilitating Quality Problems

- Complex multi-tier applications
- Legacy application integration
- Less time
- Fewer resources

Optimize Software Quality with SilkPerformer by Gaining Visibility Into Performance, Scalability and Reliability

Easily Customize Scripts Exactly To Your Needs

Learn how you can improve your quality optimization with Segue
Solutions, distributed by Metastore in Benelux. Contact us for more information.
Mithun Training & Consulting B.V.

Our Promise
Mithun Training & Consulting helps organizations optimize their resources and improve their performance by concentrating on the most important element, their ability to deliver.

What We Do
Mithun Training & Consulting is more than just a training company. We provide skill development that is relevant to your business needs. We apply our knowledge in industry models for improving processes to help organizations develop and manage software and systems. Our experts work closely with you to provide complete training and mentoring programs, helping individuals and organizations achieve their career and business goals.

Requirements Management & Engineering for outsourcing and off-shoring of ICT activities.
Many organisations are in the process of changing their ICT strategy from internal development to outsourcing and off-shoring of their ICT activities. This implies that the actual development and implementation of ICT systems will migrate to external parties abroad. By doing so, these organisations intend to significantly reduce the cost of their ICT activities, by large scale outsourcing and off-shoring.

In order to truly benefit from outsourcing and off-shoring, it is essential that the business units are able to produce requirements that are complete, correct (unambiguous and SMART) and consistent, to reduce the risk that the external party will build the wrong system.

Changing an organisation so dramatically, will have a direct impact on your employees. People will feel insecure about their job and role in the organisation. Experience has taught us that investment in people through training will reduce these feelings of discomfort, to support the staff to be able to better adapt to their new roles.

We will assist your staff to be able to find, capture, analyse, document and engineer these requirements, using a natural language, and make these requirements measurable and testable, complemented with additional modelling techniques where required.

Our core areas of expertise are:
- Requirements Management & Engineering
- Object Oriented Analysis & Design with UML 2.0
- Real-time and Embedded Analysis & Design
- Software Engineering Processes (Agile & DSDM)

We invite you to visit us at our booth at the VVSS2005 vendor show or contact us directly at:

Mithun Training & Consulting B.V.
P.O. Box 898
3800 AW Amersfoort
Netherlands
T. +31 (0) 33-457 0840
F. +31 (0) 33-457 0839
E. info@mithun.nl
I. www.mithun.nl
ps_testware as Independent Partner in Software Quality

Problems with the quality of your software? Need some help in structuring your requirements or test activities? Want information on what structured software testing really involves, what validation can mean for you and how to implement it? ps_testware helps to find a solution as Quality is (y)our business.

ps_testware is a Belgian/Dutch consultancy firm with the head office in Leuven (BE) and subsidiary in Gorinchem (NL) and we make Software Quality (structured software testing and computer system validation) our business since more than 13 years now.

Ps_testware has about 60 consultants working in the Netherlands, Belgium and France to improve the quality of software for mostly large enterprises (particularly in the banking, insurance, pharmaceutical and energy sector). These consultants are all ISEB certified and use a proven methodology (based on our Implementation (or I-)Model). They strive for qualitative software by implementing a structured test and validation process and thereby gaining precious time and money.

ps_testware is only satisfied when the customer is satisfied. What the customer wishes is what we want, resulting in a shared target: to deliver a qualitative software product through a structured and repeatable process. ps_testware delivers test and validation services in various forms: consultancy, coaching, training, co-ordination, management and outsourcing. Our methodology can be used for both manual processes and automated tests (regression testing and performance testing).

To support a structured way of working, ps_testware uses the web tool QMX (Quality Management eXpert).

QMX provides all information needed to manage and follow-up your test process:
- a clear view on the quality of the test process
- automated reporting
- linked information in an easy and synoptic way
- tracked information throughout all phases of the process
- basis for founded (release) decisions.

This test management tool was created by ps_testware based on a proven test methodology and 10 years of practical project experience. QMX is recognised as an Innovative Product and can count on the support by IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders), which is the only Flemish organisation stimulating and supporting innovation. Quality Management eXpert is currently used at several customer projects.

For more information: www.pstestware.com and www.myqmx.com
Established in 1986, Programming Research is recognized worldwide as the leading authority in the assessment of software quality and coding standard compliance through automated source code analysis and process improvements and requirements solutions provider. Products are:

- IRqA®, Requirements engineering;
- QA C MISRA;
- QA C++.

**IRqA® competitive advantages**

- IRqA® is Requirements Engineering oriented (vs Requirements Management only): The complete specification cycle is supported via standard models:
  - Requirements Capture
  - Requirements Analysis
  - System Specification building
  - Specification validation (specification vs requirements)
  - Acceptance Tests management
  - Requirements Organization & Classification
  - Requirements Management
- Provides a powerful set of modelling capabilities.
- Graphical organization and navigation model supported. Those graphical models provide key benefits over textual capture of systems structure:
  - More intuitive and flexible than "folders" approach
  - Active diagrams, not just pictures
  - Multiple Organization and Classification Layers
  - Form the basis of RM in complex consortia co-ordination
- With IRqA®, you can implement your RE management process.
- Open RDBMS-based repository (any commercial RDBMS can be used).
- Both classical functional and O.O. approaches are supported for requirements analysis and specification building.
- Provides a powerful XMI interface with XMI-compliant design tools.
- Advanced Reports Generation & Management: both specific standard-based and user-defined documents can be generated or captured.
- IRqA® provides a cost estimation module based in the concept of "Use Case-Point".
- Complexity metrics.
- Full traceability from an element to another one of any type (i.e: requirement, concept, service, test scenario, etc).

**QA·MISRA** is recognized worldwide as the leading, most powerful, and most widely adopted solution for MISRA compliance available today. QA-MISRA automatically enforces the latest MISRA guidelines now and gives you a head-start to comply with the new SAE J2632 guidelines underway for tomorrow. Our QA-MISRA Metrics Module delivers even more value by computing and reporting all statically determinable metrics found in MISRA Report 5, "Software Metrics". This report identifies software attributes and metrics which are used to measure code quality. QA-MISRA provides an efficient, practical solution to the challenge of enforcing the MISRA standard. Today, we deliver automatic enforcement of a remarkable, unrivaled 98% of the statically enforceable MISRA rules.

**QA-MISRA Features**

- Detects and reports violations of the MISRA rules
- Computes & reports all statically determinable metrics found in MISRA Report 5
- Links warning messages directly with the text of the appropriate rule
- Provides cross references via further HTML links to the appropriate rule definition and explanatory examples
- Produces code quality reports which tabulate by rule, the number of violations found in each file while linking them to the appropriate part of the source code

**QA-MISRA Benefits**

- Allows tailoring and extension of the rules to meet local requirements
- Educates developers with regard to "safe" language usage and MISRA C
- Offers an automatic, repeatable and efficient code verification method
- Establishes a software quality benchmark against which subsequent revisions of code can be measured and compared
- Provides all the standard features of the powerful QA-C environment including metrics, code visualization, demographics, and more.
Refis houdt zich bezig met de kwaliteit van geautomatiseerde systemen in de meest brede zin van het woord. Of het nu gaat om betrouwbaarheidsanalyses, metrieken en meetsystemen, of het testen van informatiesystemen, Refis adviseert, ontwikkelt en participeert in alle aspecten van kwaliteitsmeting en -verbetering.

**Onderzoek**
In nauwe samenwerking met universiteiten, opdrachtgevers en collega bedrijven werkt Refis aan een brede inzetbaarheid van bestaande betrouwbaarheidsmodellen, nieuwe hulpmiddelen en een verspreiding van kennis en ervaring.

**Advies**

"Bespaar kosten in ontwikkeling, testen en exploitatie".

**Ontwikkeling**
In nauwe samenwerking met haar opdrachtgevers, ontwikkelt en implementeert Refis meetsystemen waarmee de opdrachtgever continue inzicht heeft in de performance van de eigen auto-matiseringsprocessen en -producten.

**Opleiding**
Refis trainingen op het gebied van testen, kwaliteitszorg en systeembetrouwbaarheid onderscheiden zich door het praktische karakter en de directe toepasbaarheid van de lesstof.

"Verhoog de betrouwbaarheid van informatie- en procesbesturingssystemen".
Sogeti Nederland B.V. meer dan 30 jaar ICT-kennis en -expertise in één bedrijf. Zij maakt onderdeel uit van een internationaal netwerk van Sogeti bedrijven (ruim 15.000 medewerkers) en behoort tot de Capgemini-groep (met zo'n 60.000 medewerkers wereldwijd). Het ontwerpen, realiseren, implementeren, testen en beheren van ICT-oplossingen behoort tot haar core-business.

Software Control is een divisie van Sogeti Nederland B.V. Als eigenaar van de TMap®-methodiek en het TPI®-model is Software Control met haar 400 specialisten een trendsetter op het gebied van testen en quality assurance binnen het ICT-werkveld. De dienstverlening van Software Control concentreert zich rond requirements lifecycle management, gestructureerd testen, quality assurance en (test)procesverbetering. De vorm van deze dienstverlening varieert van detachering via testprojecten tot volledige uitbesteding van het testproces aan de TMap®Factory en met offshore-mogelijkheden bij Sogeti India. De opdrachtgevers van Software Control zijn te vinden in alle segmenten van het Nederlandse bedrijfsleven en de overheid.

Om met haar dienstverlening voortdurend de ontwikkelingen en trends in de ICT-wereld te volgen, investeert Software Control veel in research & development. Innovaties in de technologie, nieuwe ontwikkelmethoden, nieuwe toepassinggebieden en trendswijzigingen in ICT-beleid van toonaangevende ondernemingen worden op de voet gevolgd. De resultaten van research & development worden gepubliceerd in (inmiddels 14!) boeken, in de vakbladen en in (internationale) newsletters en gepresenteerd op TMap® Test Topics seminars en de nationale en internationale (test)platforms als Testnet, SPIder en Eurostar. Ook op www.tmap.net worden de resultaten in detail gepubliceerd.

Wij stellen de hoogste eisen aan onze professionals. Zij zijn dan ook zonder uitzondering van zeer hoog niveau en hebben uitgebreide training in onze dienstverlening gekregen. Via reguliere cursussen en bijeenkomsten blijven zij op de hoogte van de laatste vakontwikkelingen. Bijzonder is ook dat zij hun vak hebben gemaakt van onze dienstverlening en u dus echte specialisten over de vloer krijgt.

Software Control is gecertificeerd conform ISO 9001:2000

Sogeti Nederland B.V.
Divisie Software Control
Hoofdweg 204
3067 GJ Rotterdam
www.sogeti.nl
www.tmap.net
Software is playing an increasingly larger and more significant role in almost every aspect of our lives. Many of the devices that we use on a daily basis are (becoming) entirely dependent on software, from mobile phones through to automobiles, from DVD players to central heating controllers. The same is true within industry, from medical systems to manufacturing equipment, from process control to transport and logistic systems.

The result of this is an explosion in the size and complexity of software systems. Figures from leading companies, including Philips and BMW, show that the size of (embedded) software systems is increasing at an exponential rate, paralleling Moore’s Law. They also show that software complexity is increasing at an even faster rate.

Conversely, productivity studies show that over the last 10 years software developers have scarcely been able to keep up with demand. Figures from leading institutes, such as the SEI and QSM, show that on average the best (embedded) software development organizations – those that have a software process improvement program – have only doubled their productivity during this period; average organizations have achieved much less.

The result is a capability gap between the demand for ever larger and more complex (embedded) software systems and the average development organization’s productivity.

This gap is the source of enormous tension in the market.

Embedded software development projects are so essential to many new products that they are rarely ever allowed to fail. Instead, organizations pump money into them until some measure of success is achieved. The result is that software development projects often run massively over cost, extend the end product’s time-to-market and/or ultimately deliver a poor quality solution. For example there are an increasing number of stories about the decreasing reliability of automobiles, with some manufacturers having already suffered market share losses as a result. Predictions for the future indicate that this situation will continue to deteriorate across all markets.

A solution to these problems can only be achieved by a quantum leap – an innovation – in software development efficiency, which suggests the need for a fundamental change to the way in which software is currently developed.

Verum has developed and adapted a series of innovative mathematical software design techniques, cumulatively referred to as “Analytical Software Design” (ASD). At the heart of this technology lies a new unique mathematical method for which Verum has applied for Patent Protection. ASD is capable of bringing mathematical rigour to the process of designing behaviourally complex software systems. It is also able to increase the effectiveness of and give statistical meaning to software testing.

The application of ASD to software development establishes mathematical completeness and correctness in the specification and design phases of a project. The result is a greatly increased level of precision and a dramatic reduction in defects extremely early in the development lifecycle. The repercussions of this are felt through the entire rest of the lifecycle: Development effort can be reduced by as much as 30%, development timescales may be reduced by 30% and the number of defects in the software at delivery is reduced by 90%. Overall the effect of ASD is to increase the end-to-end predictability of software development in terms of Cost, Quality and Time-to-Market.

Please attend the presentation of Verum’s CTO Guy H. Broadfoot, called “Meeting the quality challenge of untestable software” at 15:45 or visit our booth at the Tool exhibition for more information.

www.verum.com