EINDHOVEN UNIVERSITY OF TECHNOLOGY

7e Nederlandse testdag, Eindhoven, 8 November 2001 : proceedings

Citation for published version (APA):
Feijs, L. M. G., Mauw, S., Goga, N., & Willemse, T. A. C. (editors) (2001). *7e Nederlandse testdag, Eindhoven, 8 November 2001 : proceedings*. (Computer science reports; Vol. 0110). Technische Universiteit Eindhoven.

Document status and date:
Gepubliceerd: 01/01/2001

Document Version:
Uitgevers PDF, ook bekend als Version of Record

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Aug. 2022

TU/e technische universiteit eindhoven

CS-Report 01-10

7e Nederlandse Testdag

L.M.G. Feijs
N. Goga
S. Mauw
T.A.C. Willemse

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

7e Nederlandse Testdag

by

L.M.G. Feijs, N. Goga, S. Mauw and T.A.C. Willemse (Eds.)

01/10

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/inf/onderzoek

Computer Science Reports 01/10
Eindhoven, October 2001

7e Nederlandse Testdag

Preface

These are the proceedings of the seventh edition of the *Nederlandse Testdag* (a.k.a. *Dutch Testing Day*), held on November 8, 2001 in Eindhoven, The Netherlands.

The increase in the complexity of software and hardware systems was the predominant concern in the software design of the last decades. This increase is still going on today, and mastering this complexity is possible, only by investigating, discussing and evaluating methods and techniques for testing such systems. The *Nederlandse Testdag* serves as a forum in which researchers from the industry and the academia discuss and present their latest experiences and theories in the area of testing. The initiative for organising the *Nederlandse Testdag* is, and has always been, the result of the combined efforts of the Dutch academia and the industry. The *Nederlandse Testdag* is an annual event which was first held in 1995.

This year's edition again consists of one invited presentation by Jens Grabowski, on TTCN-3, and six regular presentations, both from the academia and from the industry. The presentations capture a broad field of the entire testing spectrum. In the presentation by Martin Gijsen (CMG), test automation for Graphical User Interface (GUI), dedicated and embedded systems according to the TestFrame methodology is explained. Klaas Mateboer (Collis) presents the test-tool Conclusion. René de Vries (University of Twente) reports on specification testing in practice and illustrates this by means of an example. In the presentation by Loe Feijs (Eindhoven University of Technology), testing is related to game-theory. Marcel Verhoef (Chess) and Bertil Oving (NLR) present their experiences using real-time simulation, UML and VDM to obtain more reliable spacecraft avionics. Finally, Ben van Buitenen (Baan), provides an insight in service pack testing: how to efficiently test customised software components and packages.

The organisation of the *Nederlandse Testdag* is grateful for the sponsorship it received from the Eindhoven University of Technology, the Eindhoven Embedded Systems Institute, and the financial support from Dutch Research School IPA. We are very much indebted to CMG and Telelogic's willingness to sponsor this event financially. Over the years, both companies have profiled themselves as companies investing both time and resources in advancing the current state in testing. Finally, the organisation thanks Marcella de Rooij and Elize Russell for their organisational assistance.

Loe Feijs
Niculae Goga
Sjouke Mauw
Tim Willemse

November 2001
Eindhoven
The Netherlands

TU/e CMG Information Technology Telelogic

Contents

Putting TTCN-3 into Practice *Jens Grabowski (Institute for Telematics, University of Lübeck)*	1
Effective Test Automation for GUI, non-GUI and Embedded Systems *Martin Gijsen (CMG)*	22
Object Oriented Testing with Conclusion *Klaas Mateboer (Collis)*	42
Specification Based Testing: Lessons from Practical Applications *René de Vries (University of Twente)*	52
Prisoner's Dillema in Software Testing *Loe Feijs (Eindhoven University of Technology)*	65
Efficient Development, Verification and Validation of Spacecraft Avionics *Marcel Verhoef (Chess) and Bertil Oving (NLR)*	81
Service Pack Testing in a Commercial Development Environment *Ben van Buitenen (Baan)*	94
Information on CMG and Telelogic	111

Putting TTCN-3 into Practice

Jens Grabowski
Institute for Telematics,
University of Lübeck

©Copyright J. Grabowski, 2001

The abbreviation TTCN

TTCN **was** an abbreviation for

~~Tree~~ and **T**~~abular~~ **C**~~ombined~~ **N**otation

TTCN is **now** an abbreviation for

Testing and **T**est **C**ontrol **N**otation

©Copyright J. Grabowski, 2001

Outline

- What is TTCN-3 ?
- Concepts
- TTCN-3 Core Notation
- Graphical Presentation Format for TTCN-3
- First Experiences
- Summary and Outlook

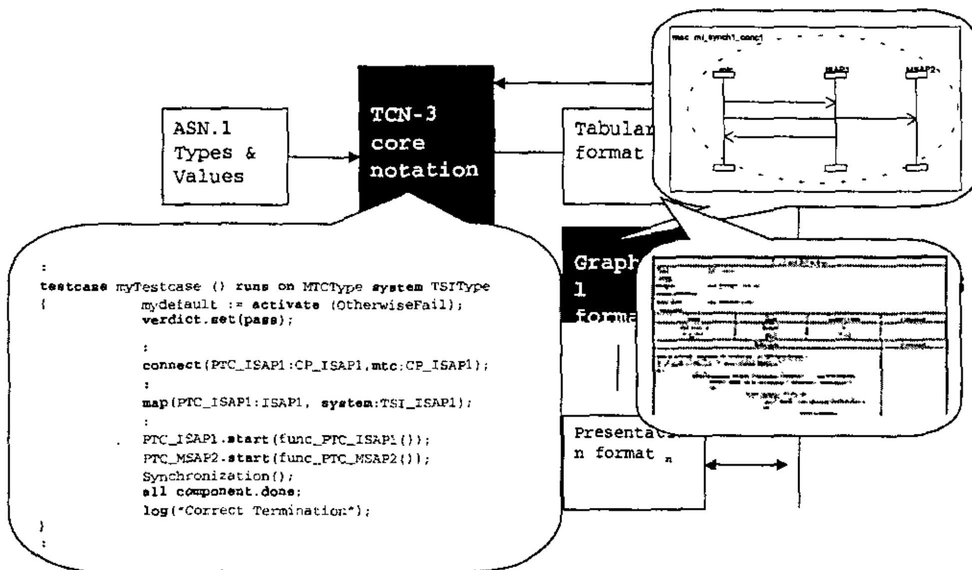
What is TTCN-3 ?

- The new standardised test specification and test implementation language
- Developed based on experiences from previous TTCN editions
  - Removal of OSI specific concepts,
  - Improvement of concepts,
  - Introduction of new concepts
- Applicable for all kinds of black-box testing for reactive and distributed systems, e.g.,
  - Telecom systems (ISDN, ATM); Mobile systems (GSM, UMTS); Internet (has been applied to IPv6); CORBA based systems.

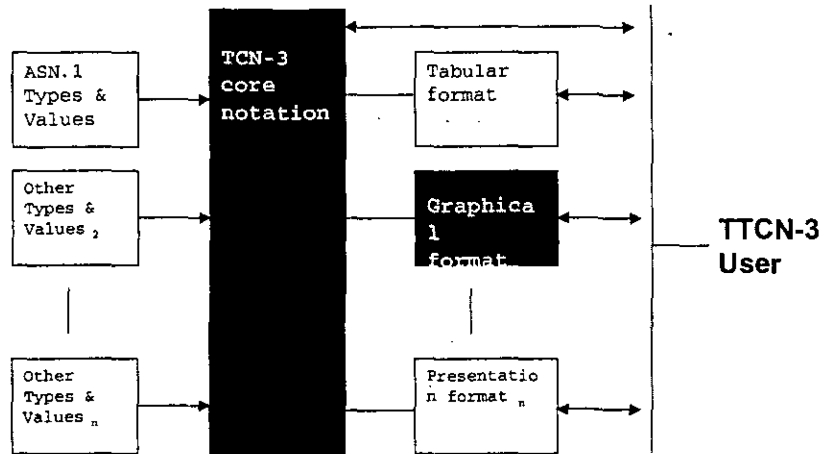
Main New Aspects of TTCN-3

- Triple C
  - Configuration: Dynamic concurrent test configurations
  - Communication: Message- and procedure-based communication
  - Control: Test case execution and selection mechanisms
- Improved
  - Harmonized with ASN.1 (and IDL)
  - Module concept
- Extendibility
  - Attributes, external function, external data
- Well-defined syntax, static & operational semantics
- Different presentation formats

What is TTCN-3 ?



What is TTCN-3 ?

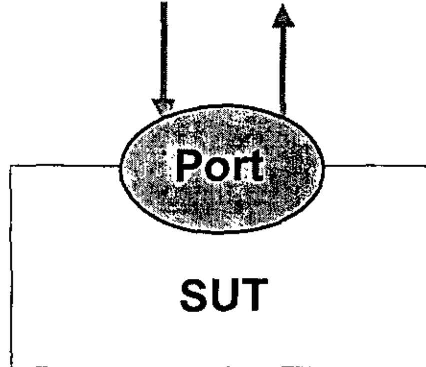


Outline

- What is TTCN-3 ?
- **Concepts**
- TTCN-3 Core Notation
- Graphical Presentation Format for TTCN-3
- First Experiences
- Summary and Outlook

Black-Box Testing

Port.send(Stimulus) **Port.receive(Response)**

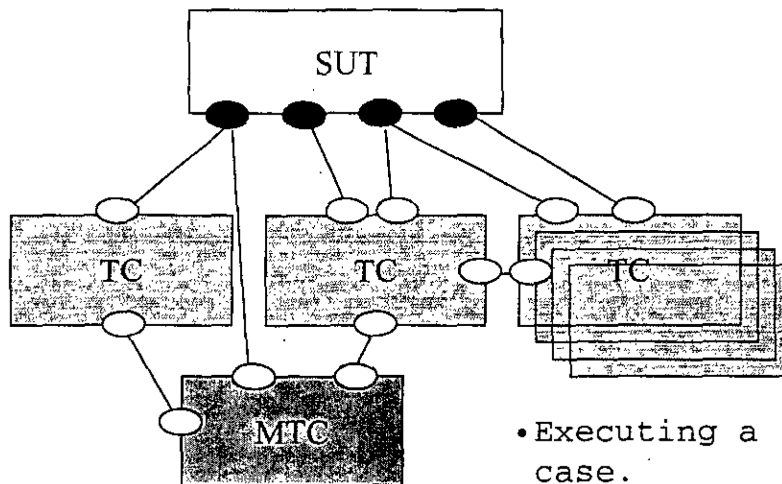


- Stimulate SUT.
- Compare observed response with expected response.
- Assign verdict.

SUT: System Under Test
Port: Formally specified system interface

©Copyright J. Grabowski, 2001

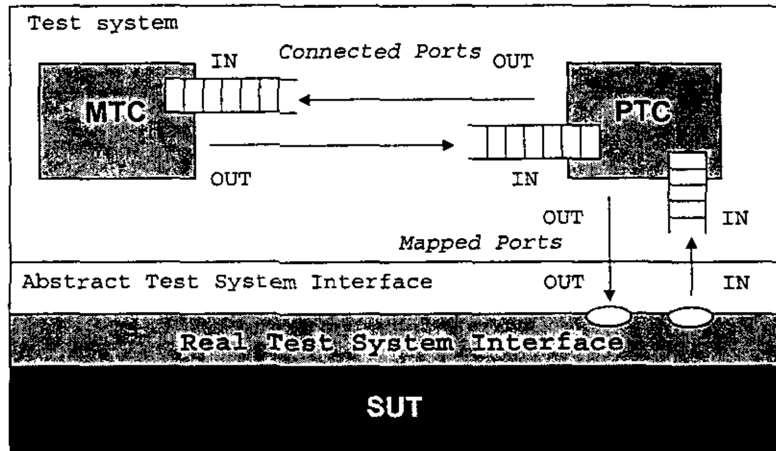
Test configuration (1)



- Executing a test case.
- Returning a verdict.

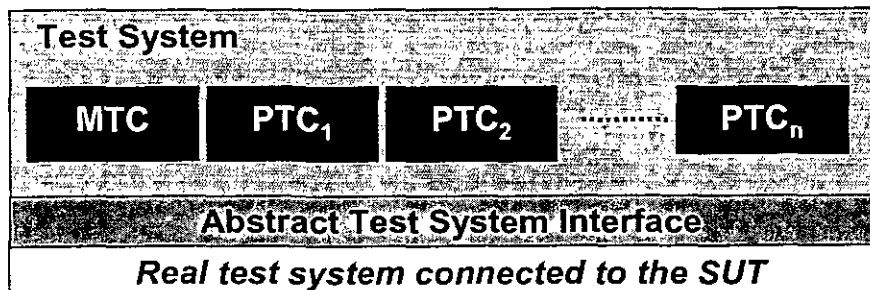
©Copyright J. Grabowski, 2001

Test configuration (2)



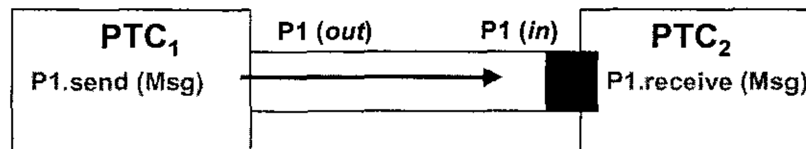
Test Components

- There are three 'kinds' of components
  - MTC (Main Test Component)
  - PTC (Parallel Test Component)
  - Abstract Test System Interface defined as component



Communication Ports

- Test components communicate via ports
- A test port is modeled as an infinite FIFO queue
- Ports have direction (in, out, inout)
- There are three types of port
  - message-based, procedure-based or mixed

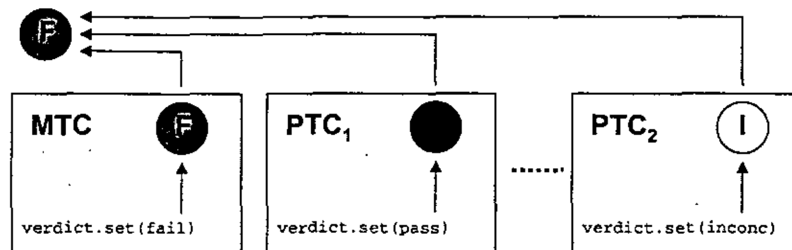


©Copyright J. Grabowski, 2001

Verdicts

- Verdicts: pass, fail, inconc, none, error
- Each test component has its own local verdict
  - can be written (set) and read (get)
- Global verdict returned by Test Case

Verdict returned by the test case when it terminates

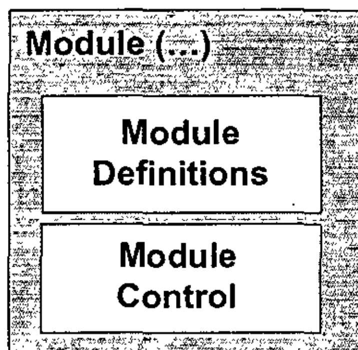


©Copyright J. Grabowski, 2001

Outline

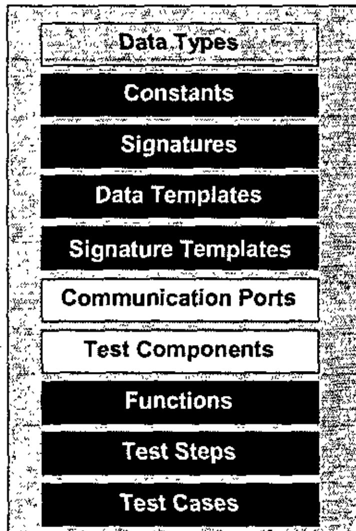
- What is TTCN-3 ?
- Concepts
- **TTCN-3 Core Notation**
- Graphical Presentation Format for TTCN-3
- First Experiences
- Summary and Outlook

TTCN-3 Modules



- Modules are the building blocks of all TTCN-3 specifications
- A test suite is a module
- A module has a definitions part and a control part
- Modules can be parameterised
- Modules can import definitions from other modules

Module Definitions



- Definitions are global to the entire module
- Data Type definitions are based on TTCN-3 predefined and structured types
- Templates define the test data
- Ports and Components are used in Test Configurations
- Functions, Test Steps and Test Cases define behaviour

©Copyright J. Grabowski, 2001

Data Type and Template Defintions

```
type record Request          {
   RequestLine               requestLine,
   ReqMessageHeader          reqMessageHeader optional,
   charstring                     crlf,
   charstring                     messageBody optional
}


template Request Invite_s_1 := {
   requestLine               := Request_Line_s_1("INVITE"),
   reqMessageHeader          :=
   Req_Mes_Header_s_1("INVITE"),
   crlf                      := CRLF,
   messageBody               := omit
}
```

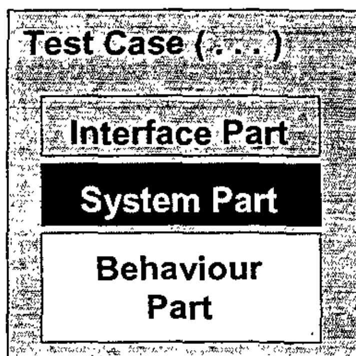
©Copyright J. Grabowski, 2001

Port and Component Type Definitions

```
type port SipPortType {
    inout Request, Response;
}


type component SipTestComponent {
    var integer Counter := 0;
    timer T1 := 0.5;
    timer T2 := 4.0;
    port SipPortType SIP_PCO
}
```


Test Case Definition



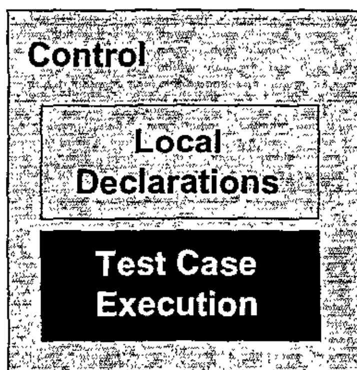
- Test cases are a special kind of function executed in the control part of a module
- The interface part (runs on) references the type of the MTC
- The system part (system) references the type of the test system interface
- The *Behaviour part defines* the behavior of the MTC

A Test Case example

```
testcase SIP_UA_REC_V_001()
 runs on SipTestComponent system SipConfiguration {
        var default mydefault := activate (Default_1("0"));
        map(self:SIP_PCO, system:SIP_PCO);
        SIP_PCO.send(Invite_s_1);
        T1.start;
        SIP_PCO.receive(Response_r_1);
        verdict.set(pass);
        T1.stop;
        deactivate (mydefault);
        postamble("0");
}
```

©Copyright J. Grabowski, 2001

Module Control



- Module control is the 'dynamic' part of a TTCN-3 specification where test cases are executed (execute)
- Local declarations, such as variables and timers may be made in the control part
- Basic programming statements may be used to select and control the execution of the test cases

©Copyright J. Grabowski, 2001

Control Part

```
control {
   var integer count := 0;
   var verdicttype myVerdict := pass;

   if ( execute ( SIP_UA_REC_V_01() ) == pass) {

         while ( count <= 10 ) {
                 myVerdict := execute(SIP_UA_REC_V_02());
                 count := count + 1;
         } // end while
   } // end if
} // end control
```

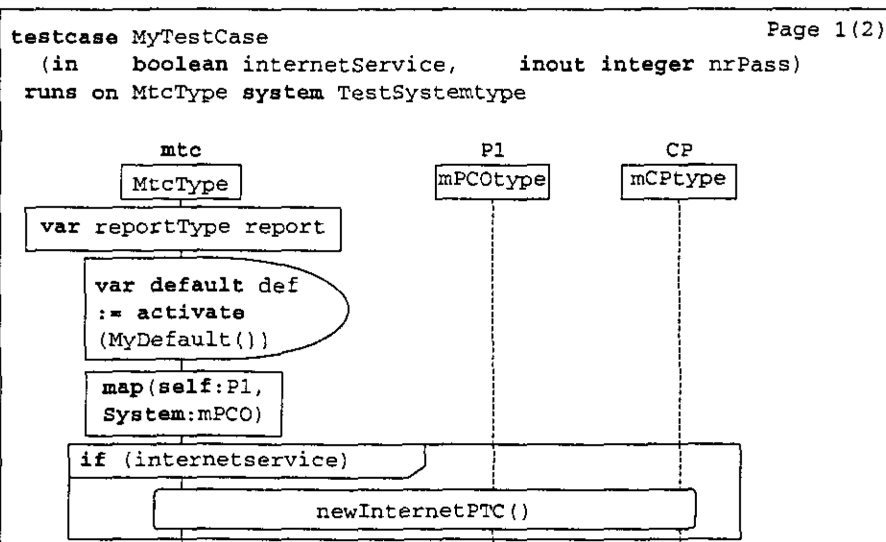

Outline

- What is TTCN-3 ?
- Concepts
- TTCN-3 Core Notation
- **Graphical Presentation Format for TTCN-3**
- First Experiences
- Summary and Outlook

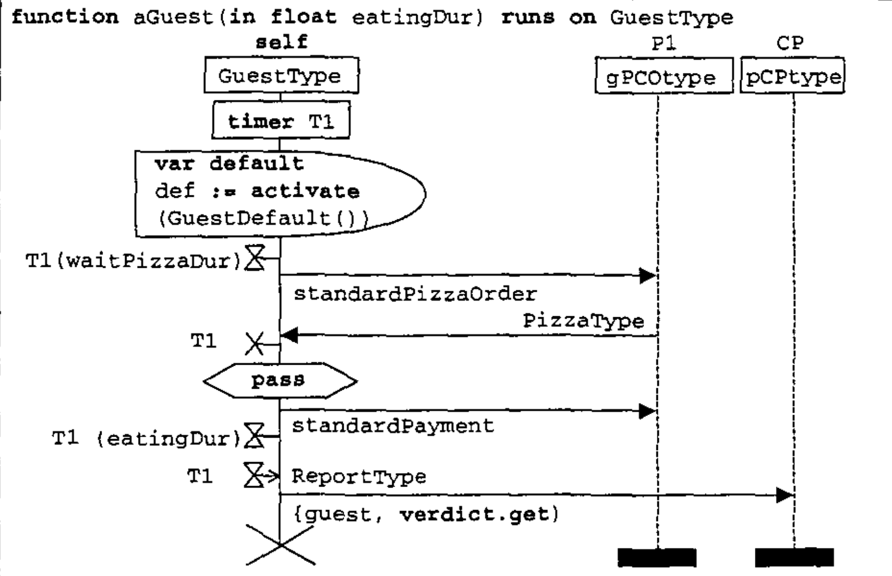
Graphical Presentation Format (GFT)

- GFT is based on MSC-2000
- GFT uses the TTCN-3 data descriptions
- GFT defines several MSC extensions to make MSC-2000 applicable in the testing context
- In GFT each TTCN-3 test case, function and test step is presented in form of a GFT diagram
- GFT defines only the requirements for a graphical presentation of the module structure but no concrete graphics

GFT Examples (1)

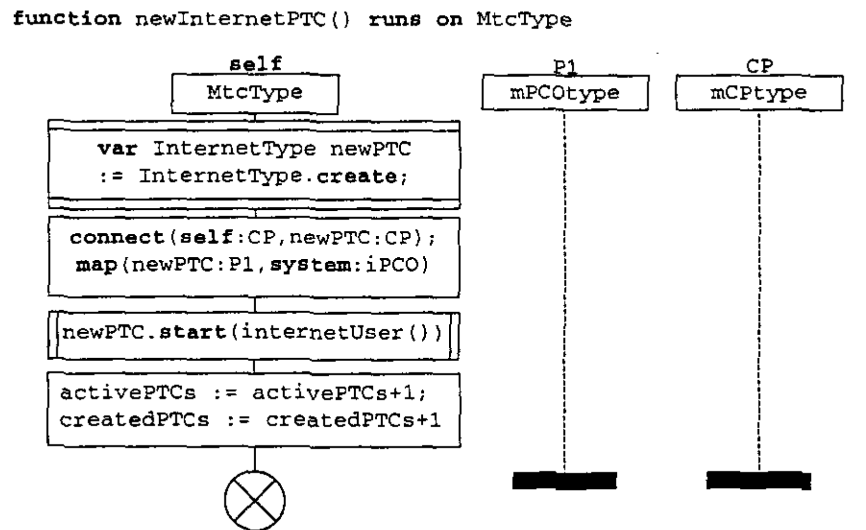


GFT Examples (2)



©Copyright J. Grabowski, 2001

GFT Examples (3)



©Copyright J. Grabowski, 2001

Outline

- What is TTCN-3 ?
- Concepts
- TTCN-3 Core Notation
- Graphical Presentation Format for TTCN-3
- **First Experiences**
- Summary and Outlook

First Experiences (1)

- TTCN-3 has been developed 1999-2001
- The first commercial tools are available or will be available at the end of 2001
  Telelogic (Sweden), Testing Technologies (Germany),
  Danet (Germany), Da Vinci Communications (New Zealand)
- Internal work on TTCN by
  Nokia (Germany, Finland), Ericsson (Sweden, Hungary)
  Nortel (Canada), Motorola (UK, China)
- Academic tools
  University of Lübeck, Technical University of Berlin

First Experiences (2)

- Trial applications
  - *IPv6 testing*
    - performed by Ericsson (Hungary)
  - *Session Initiation Protocol* (SIP)
    - performed by ETSI (France) and GMD (Berlin)
- Strong commitment of the UMTS testing group to use TTCN-3 in 2001 if the tools are available

©Copyright J. Grabowski, 2001

Outline

- What is TTCN-3 ?
- Concepts
- TTCN-3 Core Notation
- Graphical Presentation Format for TTCN-3
- First Experiences
- **Summary and Outlook**

©Copyright J. Grabowski, 2001

Summary

- New version of the only standardized test notation
- TTCN is widely used and well-established in the telecom domain
- Programming-like testing language with flexible data support and several presentation formats
- Wider scope of application
  - applicable to many kinds of testing and not just conformance testing (development, system, integration, interoperability, scalability ...)
  - applicable in the datacom domain
- Harmonization
  - first choice for test specifiers, implementors and users both for standardized test suites and
  - as a generic solution in industrial software development

Kinds of testing TTCN-3 can do
- by Nortel -

- Conformance
- Interoperability
- Configuration
- Compatibility
- Performance
- Stress
- Robustness
- Integration
- Functional
- Load
- Reliability
- Fault tolerance
- Scalability
- Degraded mode
- Unit
- Product
- Development
- Design
- Interface
- System

Outlook

- Maintenance of TTCN-3
  - Corrections of the Core Language at the End of 2001
  - Intensive work on the Graphical Presentation Format
- TTCN-3 runtime interface is under development
- New TTCN-3 features under discussion
  - Regular expressions (planned for Dec. 2001)
  - Global timer (e.g. to express hard real-time requirements)
  - Concurrent execution of test cases

GFT Ongoing Work

- Completion of GFT in terms of
  - Language concepts.
  - Grammar definition.
  - Mapping from and to TTCN-3 Core Notation.
- Convergence with MSC-2000.
- Case studies on the use of GFT.
- Transition to UML
  - UML is lacking test specification support.
  - MSC-2000 is proposed as input to UML v2.0.
  - GFT could become a UML testing profile.

Thank you for your attention.

©Copyright J. Grabowski, 2001

Who is making TTCN-3

- Individuals
  - Anthony Wiles (Project Leader, ETSI)
  - Colin Willcock (Nokia),
    Jens Grabowski (ITM Lübeck),
    Ina Schieferdecker (FOKUS),
    Ekkart Rudolph (TU München),
    Paul Baker (Motorola),
    Johan Nordin (Telelogic)
  - Ostap Monkewich (ITU-T Rapporteur, Nortel),
    Dieter Hogrefe (ETSI MTS chairman, ITM Lübeck)
- Organizations
  - Nokia, France Telecom, Motorola, Ericsson, Nortel, Tektronix, Danet, FOKUS, TU Berlin, Telelogic, NMG Telecoms, Da Vinci Communications, Testing Tech, Fraunhofer Gesellschaft, ITM Lübeck

©Copyright J. Grabowski, 2001

Contact and Further Information

- Contact

Anthony Wiles (Anthony.Wiles@etsi.fr)
Colin Willcock (Colin.Willcock@nokia.com)
Jens Grabowski (jens@itm.mu-luebeck.de)
Ina Schieferdecker (schieferdecker@fokus.fhg.de)
Ekkart Rudolph (rudolphe@informatik.tu-muenchen.de)
Dieter Hogrefe (hogrefe@itm.mu-luebeck.de)
Paul Baker (Paul.Baker@motorola.com)
Johan Nordin (Johan.Nordin@telelogic.com)

- Further information:

www.etsi.org/ptcc/ptccttcn3.htm (TTCN online information)
publications@etsi.fr (for ETSI documents)

Bibliography

- ETSI ES 201873-1 TTCN-3: *Core Language*. 2001.
- ETSI TR 201873-3 TTCN-3: *Graphical Presentation Format* (*GFT*). 2001.
- J. Grabowski, A. Wiles, C. Willcock, D. Hogrefe: *On the Design of the new Testing Language TTCN-3*. 13th IFIP International Conference on Testing Communicating Systems' (Testcom 2000), Ottawa (Canada), Kluwer Academic Publishers, August 2000.

Bibliography

- Jens Grabowski: *TTCN-3 - A new Test Specification Language for Black-Box Testing of Distributed Systems.* 17th Intern. Conf. and Exposition on Testing Computer Software (TCS'2000), Theme: Testing Technology vs.Testers Requirements, Washington D.C., June 2000.
- E. Rudolph, I. Schieferdecker, J. Grabowski: *HyperMSC - a Graphical Representation of TTCN.* Proceedings of the 2nd Workshop of the SDL Forum, Society on SDL and MSC, Grenoble, June 2000.

Bibliography

- P. Baker, E. Rudolph, I. Schieferdecker: *Graphical Test Specification - The Graphical Format of TTCN-3.* In „SDL2001: Meeting UML", LNCS 2078, Springer, 2001.
- I. Schieferdecker, S. Pietsch, T. Vassiliou-Gioles: *Systematic Testing of Internet Protocols - First Experiences in Using TTCN-3 for SIP.* Proceedings of the 5th Africom Conf. on Communication Systems, Cape Town, South Africa, May 2001.
- Paul Baker, Jens Grabowski, Ekkart Rudolph, Ina Schieferdecker. *A Message Sequence Chart-profile for Graphical Test Specification, Development and Tracing.* Proceedings of the 18th International Conference and Ex-position on Testing Computer Software, Theme: Meeting the New Challenges of Testing, Washington, D.C., June 2001.

Effective test automation for GUI, non-GUI and embedded systems

Martin Gijsen
Martin.Gijsen@cmg.nl
CMG

Presentation overview

- Preparing for test automation
- The TestFrame methodology: theory & practice
- Related topics of interest

Why automate test execution?

- Time - faster testing
- Quality - more testing
- Cost - cheaper testing
- Fun - happier testers

- These are (conflicting) business objectives
- Automating tests is not a goal in itself

What should be automated?

- What can be automated?
  - Test = initial state + stimuli + observable behavior
  - Error recovery
- What are the benefits?
  - How much time will automating save?
  - How many more tests can be performed?
- What are the costs?
  - Time to support automated test execution
  - Investing in hardware, software, licenses & training
  - Hiring consultant (for know-how / staff)

GUI vs. non-GUI test automation

GUI systems:

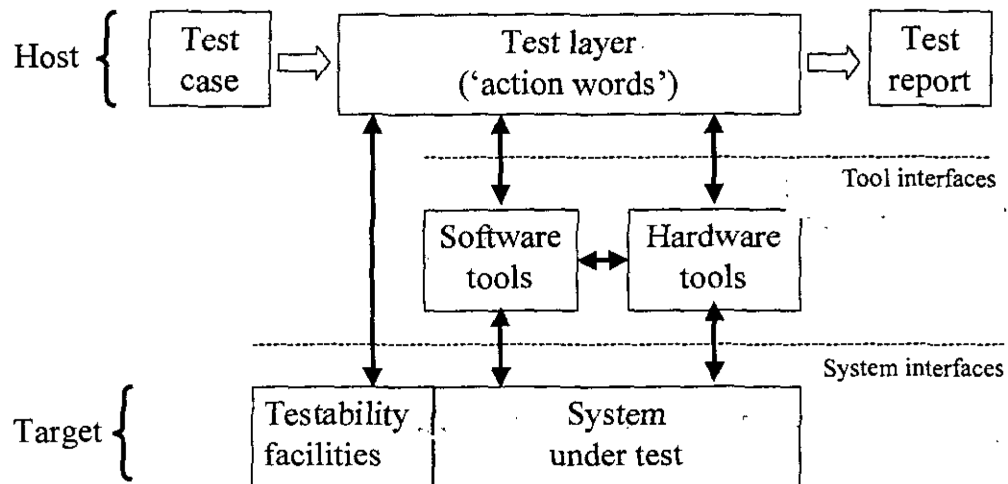
- Standard interface, off-the-shelf test tools

Non-GUI and embedded systems:

- Special software & equipment for special interfaces
- Precise timing

- Interfacing solutions will differ for different products
- Consider (automated) testability early

Interfacing with a system



Common test automation challenges

- Automated tests are hard to read and write
- Automated tests require programming skills
- Programming testers are rare
- Sensitivity to maintenance of testware

These resulted in the development of
the TestFrame methodology

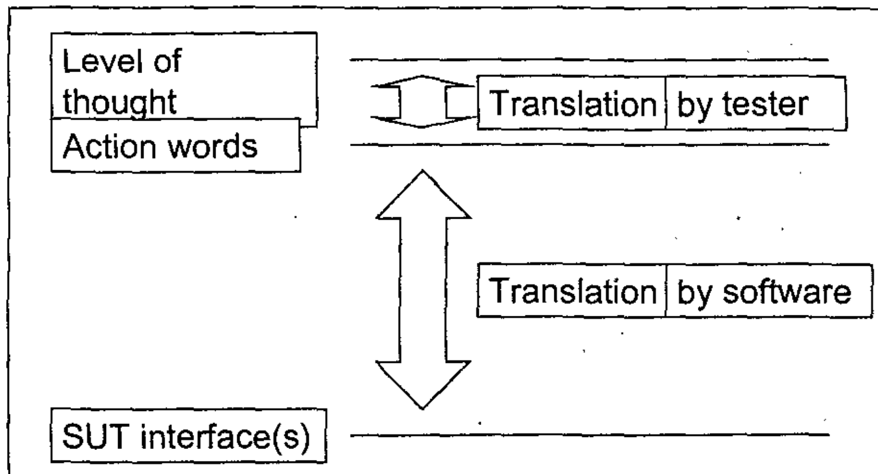
basics

The ***testers*** define 'action words'

Action words:

- the test commands
- normal words, usual vocabulary
- natural level of abstraction
- no test execution, interfacing or tool details
- no programming experience from testers required

Abstraction levels



Sample for GUI: Bank application

	name	account nr	deposit
open account	John Doe	123456789	1234
	name	number	amount
deposit cash	John Doe	123456789	1234
	name	number	amount
check account	John Doe	123456789	2468
	name	number	amount
withdraw cash	John Doe	123456789	2000
	name		
close account	John Doe		
	name		
check no account	John Doe		

- GUI system
- Host = target
- Interface: GUI
- Tool: GUI test tool

- Also works when testing through middleware!

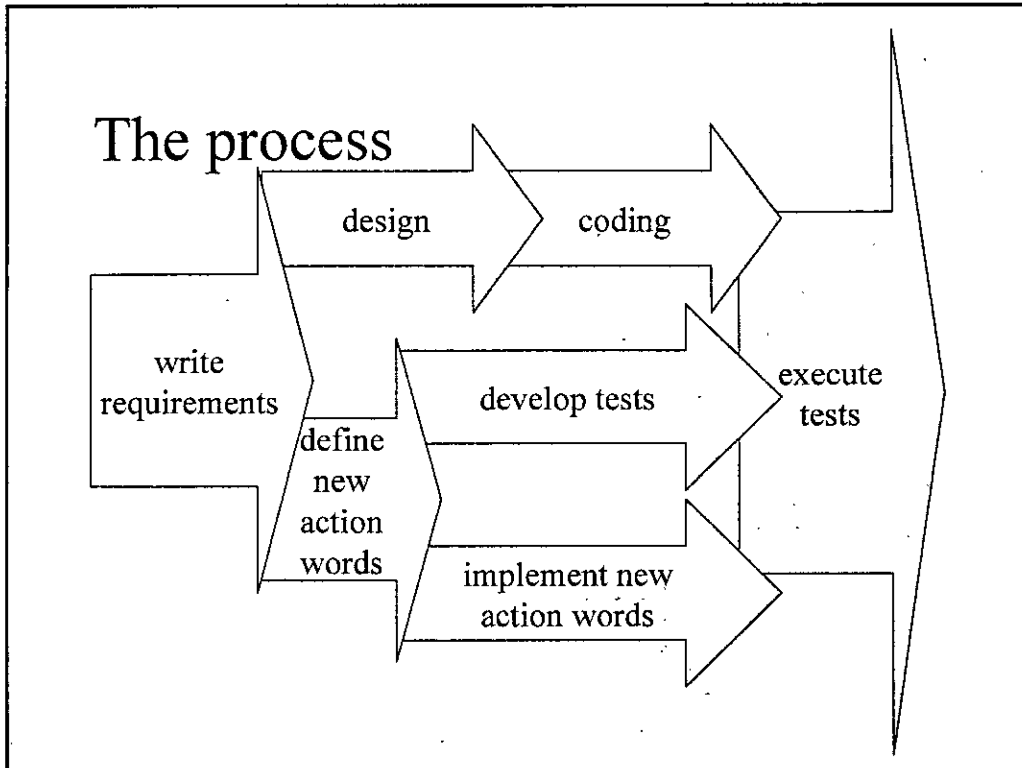
Sample for embedded: mobile phone

	code	
on	1234	
	name	number
add number	John Doe	0123456789
off		
	code	
on	1234	
	name	number
check number	John Doe	0123456789
	name	
make call	John Doe	
	name	
remove number	John Doe	
	name	
check no number	John Doe	
off		

- Embedded system
- Host ≠ target
- Interfaces: keys, display, antenna, serial port
- Testability:
  - use serial port
  - some extra SW in phone
  - most logic on host

Some key TestFrame advantages

- Action words make tests easy to read, write and maintain
- Clear separation of roles:
  - test analyst: subject expert, focus on functionality (what)
  - navigation engineer: software engineer, focus on test execution details (how: tools and interfaces)
- Reduces sensitivity to maintenance of tests and test execution platform



Some remarks

- TestFrame analysis:
  clusters & test conditions
- Software engineering effort
- Action words at lower abstraction level
- Resources
- Error recovery

Conclusions

- Business objectives guide test automation
- Non-GUI only means other interfacing
- Ensure testability in requirements phase
- Action words, for any system:
  - easy to read, write & maintain tests
  - low maintenance sensitivity
  - separate 'what' and 'how' into 2 roles

Related topics of interest

- Integrating with use cases
- Test generation from (behavioral) models:
  - Functional level of abstraction
  - Model is kept simple
- Integrating with formal methods
- Testing through middleware

Sources of information

- Soon: Martin Gijsen: white paper
- Hans Buwalda & Maartje Kasdorp: "Getting automated testing under control", STQE magazine, Nov/Dec 1999
- Mark Fewster & Dorothy Graham: "Software Test Automation – Effective use of test execution tools"
- http://www.testframe.com

Embedded TestFrame,
An Architecture for Automated Testing of Embedded Software

Harro S. Jacobs and Peter H.N. de With

CMG Eindhoven B.V. - Sector Trade, Transport & Industry, Luchthavenweg 57
P.O. Box 7089, 5605 JB Eindhoven, The Netherlands, e-mail: tswe@cmg.nl

Abstract - **Embedded TestFrame is an architecture for automated testing of embedded software. Testing is performed at two levels: 1) test specification based on spreadsheets and 2) test implementation using mature programming languages. In addition, test implementation is partitioned over a test computer and the embedded system, to minimize the overhead for the embedded system that often has limited resources. The use of mature programming languages is advantageous, because experience with and tooling for these languages is widespread. The use of spreadsheets supports an abstract test specification in an early stage without having the final interface available of the software to be tested.**

We have successfully implemented this architecture at Philips Semiconductors, where Embedded TestFrame has been accepted as the primary solution for all test activities.

Keywords - **automated testing; embedded software; architecture; host-target communication**

I. INTRODUCTION

With the advent of digital television, set-top boxes, and mobile telephones, embedded systems which were conventionally performing control only, have become so powerful that a multitude of processing tasks, including applications and user interaction, are carried out. Recent architectures for high-end digital audio and video systems contain (multiple) 32- or even 64-bit CPUs and DSPs and up to 64 MB RAM. The corresponding embedded software shows a strong complexity increase due to augmented memory size. As a result, the total development time is increasingly determined by the software development time.

Due to the complexity and size of embedded software together with strong demands on time-to-market and quality, testing is a crucial point that should be addressed during software development. Traditionally, testing is carried out during the last phases of the software development life cycle. As a consequence, testing activities are often subject to high time pressure, which either results in delayed market introduction or low product quality. Furthermore, high recall costs for embedded systems should be avoided.

In this paper, we propose a novel architecture for automated testing of embedded software, named Embedded TestFrame, featuring the possibility to start test development in an early stage. We advocate an incremental approach for test development that can already be started as soon as the first requirements are fixed. Test execution can then take place as soon as the first component[1] is developed and thus provide early feedback in case of errors. The advantage is that the effort can be spread over a longer, better manageable period.

During software development, it is advisable to re-execute tests for completed components on a regular basis, because context changes may impact components that were considered to be correct. Moreover, re-execution of tests plays a crucial role during software maintenance, where new releases should be verified thoroughly. In conclusion, many situations exist in which it is required to repeat test execution regularly. In these situations, automated testing is often cost effective. The benefit of automated tests is that they provide a rapid though very reliable and reproducable statement of the product quality. As such, repeated execution of automated tests gives a good indication of the product quality over time, offering valuable metrics for project control. For the above-mentioned reasons, automated test execution has been adopted as a key feature of Embedded TestFrame.

Development of an automated test suite must not be underestimated, because test suites often turn out to be equally large or even larger than the software to be tested. One should always be aware of the trade-off between effort and (knowledge about) product quality; one may choose to only automate tests for very critical components, and to do manual tests for the remaining system parts.

This paper is divided as follows. Section II describes important characteristics of embedded systems and embedded software and discusses required architectural elements of Embedded TestFrame. Section III presents the overall framework for automated testing. Section IV addresses the architecture of Embedded TestFrame and discusses the partitioning between host and embedded

[1] In this paper, we do not have the intention to distinguish between components, modules, etc., but use the term 'component' for any clearly defined piece of software that can be tested.

system. In Section V, we focus on an important tool in this architecture, called ActiveLink, which features seamless communication between host and software to be tested. Section VI deals with the implementation of Embedded TestFrame at Philips Semiconductors. Section VII presents the conclusions.

II. CHARACTERISTICS AND REQUIREMENTS

Prior to presenting an architecture for the automated testing of embedded software, the key characteristics of embedded software are discussed and the corresponding requirements for the architecture are mentioned.

A. *Relatively high complexity of software*

The complexity of embedded software is rapidly increasing. As mentioned before, the size of a test suite may become very large, and sometimes even exceeds the size of the software to be tested. Thus, an architecture for testing embedded software should enable a controlled and incremental development of test suites.

B. *Large variety of embedded systems*

Embedded software runs on dedicated embedded systems, which will be referred to as *targets* from now on. A large variety of targets exists given the broad choices of processors, boards, (real-time) operating systems, programming languages, development environments, etc. An architecture for the automated testing of embedded software must deal with this large variety.

C. *Resource-constrained targets*

Typically, targets have constrained resources with respect to, for example, processing power and memory size. Although Moore's law - the periodical doubling of resource capacities - also applies to the embedded domain, embedded systems are often still not 'oversized', due to small profit margins. An architecture for testing embedded software should be apt to such situations and should provide means to keep the major part of a test suite outside the target.

D. *Software interfaces*

The software interfaces of a target are the interfaces that can only be accessed by software that executes on the target itself, see Figure 1. An example of a software interface is the Application Programmers Interface (API) of the software to be tested. Other examples are those applications that provide for or absorb data of the software to be tested. An architecture for testing embedded software should enable the test suite to control these software interfaces.

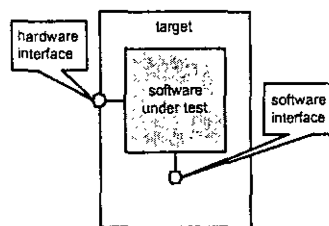


Figure 1: Software and hardware interfaces.

E. *Hardware interfaces*

The hardware interfaces are the interfaces on the physical boundaries of the target, which are controlled or observed by the embedded software, see Figure 1. Examples are serial and parallel ports, but also manual switches, LEDs, and display devices. An architecture for testing embedded software should enable the test suite to control the hardware interfaces. The architecture should not be limited to a certain set of known hardware interfaces, but it should be extensible, because the number and variety of these interfaces are continuously growing.

F. *Software reusability and portability*

Since the complexity of embedded software is increasing rapidly, components are no longer developed for a single system, but are applied in classes of systems. Therefor, reusability and portability of embedded software is of growing importance. Consequently, it must be possible to develop a test suite - or at least a large part of a test suite - that is target independent and can be used for a class of systems.

III. TESTFRAME

A. *Method*

We have developed a technique, called *TestFrame*, in order to deal with test suites of highly complex software (not specifically embedded software), see [1] and [2]. This technique makes a clear distinction between two phases: the test *specification* and the test *implementation* or *navigation*, which will be briefly explained.

Test specification - In this phase, spreadsheets are used in which high-level keywords with parameters, i.e., *action words*, are listed. These action words are domain-specific and represent an abstract definition of the test stimuli and the expected responses. The spreadsheets are based on the software requirements and do not consider the actual interfaces of the software to be tested. The test developer defines the action words and constructs the spreadsheets.

Test navigation - In this phase, the action words that have been defined during test specification, should be linked - or navigated - to the actual interfaces of the

software to be tested. Sometimes, this link is a one-to-one mapping on the interface functions of the software to be tested. However, because of the allowed abstraction in the test specification, the test navigation can be considerably large.

B. Architecture

The architecture of TestFrame is depicted in Figure 2. The figure shows the separation between the test specification and navigation, as well as the *TestFrame Engine* and the *test report*.

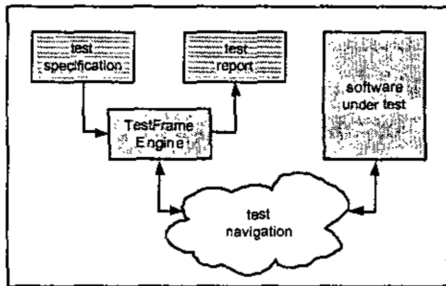


Figure 2: Architecture of TestFrame.

The TestFrame Engine is responsible for the test execution as follows. The tool sequentially parses the test specification, i.e., the spreadsheets, and communicates each action word to the test navigation. The test navigation controls and observes the software to be tested and sends the results to the TestFrame Engine. Finally, the TestFrame Engine generates a test report with a complete execution trace of the test as well as a management summary briefly showing which tests failed.

The separation between the test specification and the test navigation allows for a structured development of test suites. The test specification can already be written when the first requirements are known. Test navigation can be developed in a later stage when the software and hardware interfaces of the software to be tested have been defined. Note that the distinction between test specification and navigation also allows for specialization in the project team, e.g., analysts writing test specifications, and software developers constructing test navigation.

IV. EMBEDDED TESTFRAME

A. Architecture

Embedded TestFrame is the appliance of TestFrame in the embedded domain. As mentioned before, embedded software is typically executed on a target with limited resources. For this reason, the architecture distinguishes a test computer, i.e., the *host*, and a target, see Figure 3.

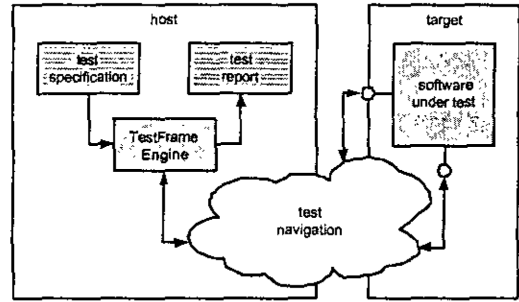


Figure 3: Host and target in Embedded TestFrame.

The host is used for storing large parts of the test suite, such as the test specification, the test report, the TestFrame Engine, and part of the test navigation. As such, the overhead on the target can be minimized.

The test navigation is responsible for connecting the TestFrame Engine and the software and hardware interfaces of the software under test. The Embedded TestFrame architecture foresees a number of modules to realize these connections, see Figure 4.

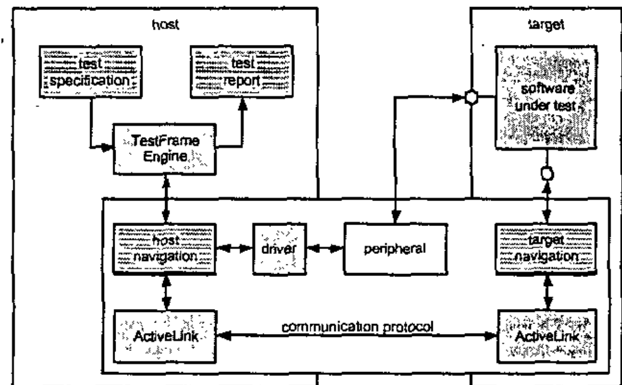


Figure 4: Architecture of Embedded TestFrame.

The most prominent module is the host navigation, which implements the action words. As such, the host navigation is domain specific and must be developed by the test engineers. This module must be developed in proven programming languages, like C, C++, or Java. We explicitly avoided the development of a dedicated script language, because knowledge of and experience with proven languages is better available and tool support is mostly mature (integrated development environments, source level debuggers, etc.).

B. Hardware interfaces

Because of the large variety of hardware interfaces, we do not strive for a library to control and observe all these interfaces. Furthermore, for some of these interfaces, such as manual switches and LEDs, dedicated hardware/software tools should be developed. Considering the effort, these are typical interfaces for

which often is chosen to abandon automated testing, and instead to control these interface manually.

For many hardware interfaces, such as serial and parallel ports, drivers and/or peripherals are available. In this case, these drivers should be integrated in the host navigation, see Figure 4. Also in this case, the use of mature languages is beneficial for it eases integration. For example, Windows drivers for serial communication can be used in a straightforward way. The strength of Embedded TestFrame is that the architecture is open en flexible for the integration of third party drivers.

C. *Software interfaces*

Since a connection must exist between the host navigation and the software interfaces of the software under test, an explicit communication means is required. Because many communication protocols (RS232, TCP/IP, JTAG, etc.) are available and proven standards for unified high-level communication are virtually absent, we developed *ActiveLink*. This tool offers a small-sized communication mechanism for transparent host-target communication at a functional level. ActiveLink eases the communication between code executing on the target and code executing on the host.

Figure 4 shows that navigation code can reside on the host as well as on the target. As discussed before, code on the target should be minimized and navigation should therefor as much as possible be implemented on the host. Although rules of thumb exist how this partitioning should take place, test developers are free to deviate and to apply a dedicated partitioning scheme. Another aspect of the partitioning is that host-target communication clearly influences the real-time behavior of the software to be tested. If this hampers testing, one should develop navigation code on the target that is critical for supporting real-time operation.

V. ACTIVELINK

A. *Architecture*

An important tool in the Embedded TestFrame architecture is ActiveLink that offers a seamless connection between host and target, while abstracting from the actual communication protocol. ActiveLink is a solution for C and C++ environments. For Java environments, we use Remote Method Invocation (RMI), a standardized Java solution that is comparable to ActiveLink.

ActiveLink offers a Remote Procedure Call (RPC) mechanism, which allows the host to call a function that is implemented on the target, and vice versa. Furthermore, ActiveLink allows to control remote memory, i.e., to dynamically allocate memory on the target and to copy memory between host and target.

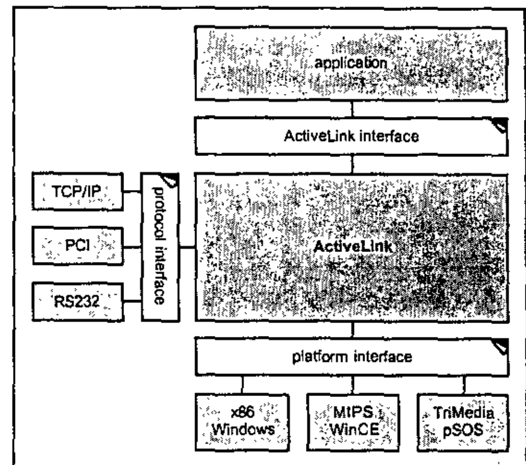


Figure 5: Architecture of ActiveLink.

Because of the large variety of targets, the architecture of ActiveLink focuses on portability, see Figure 5. The figure shows two porting interfaces: the *platform interface* and the *protocol interface*.

Platform interface - This interface abstracts from platform-specific details, such as the processor and the (real-time) operating system. For each platform, these specific details should be made available to ActiveLink, which has already been realized for Windows 95/98/NT/2000, pSOS, Linux, Solaris, and VxWorks. The platform interface enables us to port ActiveLink to other platforms with relatively little effort.

Protocol interface - This interface abstracts from the actual communication protocol between host and target, and supports already communication over TCP/IP, PCI, and RS232. The protocol interface enables extending ActiveLink with any communication protocol as long as reliable bi-directional data transfer is available.

B. *Wide applicability*

ActiveLink has not been specifically designed for Embedded TestFrame; it is a highly portable tool for cross-platform communication on application level. Therefor, it enables development of distributed applications in heterogeneous environments. It can be used for other purposes as well, like remote maintenance and control, and remote diagnostics. Recently, we realized a tracing tool based on ActiveLink to analyze the dynamic behavior of embedded software.

VI. IMPLEMENTATION AND EVALUATION

Embedded TestFrame has been implemented successfully at Philips Semiconductors within the Reuse Technology Group (RTG). This department develops reusable components for the domain of digital audio and video systems, such as digital televisions, set-top boxes (satellite receivers for digital video), and DVD players. Besides a PC-based simulation environment, RTG

currently uses MIPS- and TriMedia-based systems with the operating systems pSOS and WinCE. Also dual processor solutions executing different operating systems are being used.

In the initial phase, Embedded TestFrame was used for the automated testin g of a 2D graphics component on these systems for graphical user interfaces with menus, animations, etc. An existing test application was integrated in the Embedded TestFrame architecture (as target navigation) and spreadsheets and host navigation for additional test cases were written. ActiveLink was used for host-target communication to call the API of the 2D graphics component. Additionally, ActiveLink was used for comparing the generated bitmaps of the 2D graphics component with reference files that were stored on the PC. As such, the tests that formerly required visual inspection, were automated.

It was found that the choice for high-level languages C and C++, led to a steep learning curve for the test developers, because of their experience with these languages. The test suite was target independent and was executed on a periodical basis to test the 2D graphics component on different systems.

The successful implementation of Embedded TestFrame and its ease of use has resulted in the full integration of this package in the RTG tool set, and it is currently being used for other projects as well.

VII. Conclusions

We have presented an architecture for the automated testing of embedded software. This architecture is generic and aids to structured development of test suites. Important requirements for this architecture are that it should cope with a large variety of targets and the constrained resources of these targets.

The presented architecture offers the ability to partition tests into three parts: test specification, test navigation on host, and test navigation on target. This partitioning is highly flexible, because it needs no *a-priori* decisions about *where* to put *what* functionality.

A key feature of the Embedded TestFrame architecture is that developers can concentrate on the test functionality, while two tools, i.e., the TestFrame Engine and ActiveLink, support the partitioning and hide the platform and interface specific features.

The successful introduction of Embedded TestFrame at Philips Semiconductors has resulted in a continued development of this architecture in order to cope with new technologies. It is our intention to expand the range of targets for using Embedded TestFrame and to increase the flexibility of this solution according to the needs of our customers.

VIII. References

[1] CMG, *"TestFrame, Een Praktische Handleiding Bij Het Testen van Informatiesystemen"*, ten Hagen & Stam Uitgevers, ISBN 90-76304-67-X, Den Haag, 1999

[2] Hans Buwalda, Maartje Kasdorp, *"Getting Automated Testing Under Control"*, Software Testing & Quality Engineering, November / December 1999

Test automation for embedded and other dedicated systems

Martin Gijsen

CMG

Martin.Gijsen@cmg.nl

GUI screens are today's de facto standards for software user interfaces. It therefore makes sense that test automation, or automated test execution, is used most for GUI applications. Dedicated systems, however, often have other user interfaces or none at all. Test automation for these systems is not quite the same. But what exactly is different? And how can it be done effectively?

***

We use dedicated systems every day. Many are embedded systems, which integrate software with the hardware it controls, unlike PC software. Your mobile phone and the coffee machine in the office are some examples. Other systems are also dedicated in that they are built for one purpose, but are not really embedded. One example is an ATM, which can consist of a regular PC connected to a monitor, a card reader, a small keypad and a cash dispenser. All these dedicated systems have specific interfaces and share the complications these have for test automation.

Pacemaker

Dedicated systems can be 'mission critical' or 'safety critical', implying that system failures can have such severe consequences that they must be extremely rare. While some consider the quirks of the office coffee machine serious, they would be orders of magnitude more serious in a pacemaker or nuclear reactor. Failure is unacceptable when lives are at stake. High quality is also important because of the cost of fixing problems once the product is in the field, especially for embedded systems. Simply having a test process, even a good one, is not enough. The whole development process must carefully guard product quality. The test process then checks that the product does indeed meet its requirements.

It makes perfect sense to automate the testing of software for dedicated systems. And not just because of the higher quality-to-market, shorter time-to-market, lower cost and less repetitive work that result from effective test automation. If the timing of actions has to be more precise than is possible manually, automation is the only way. But even when the timing does not

have to be that accurate, testing dedicated systems often requires special tooling. These tools often include both hardware and software. Examples are the hardware that supports wireless communication with the mobile and the software to control that hardware from a host computer. Using these tools to automate test execution is a logical next step. But is it always possible?

'Back door'

Testing requires providing a system with certain stimuli and checking its responses. A test action for a mobile could be pressing a button or sending it a certain message. The text on the display and a phone number that is sent out are system responses. All of this takes place at the system interfaces, in this case the buttons, display and antenna. If these interfaces do not support an action or check, some tests cannot be executed. To automate testing, actions must be possible through automated means. This includes bringing the system in the initial state for the test case.

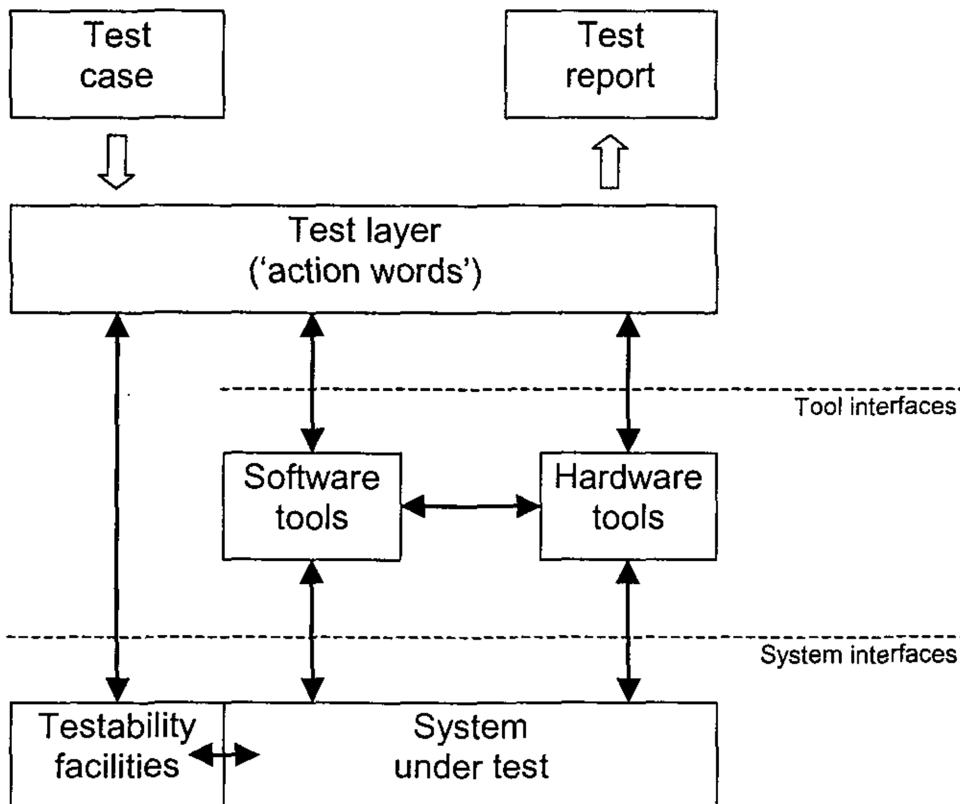


figure 1: The test environment, including product interface enhancements for testing

Automated test execution is a co-ordinated effort. It involves the system under test, the tools that talk to it and preferably an additional layer of software that addresses the tools (see figure 1). We will return to the importance of this layer shortly. The limitations to test automation

are usually in the system itself or in the tooling. How are mobile buttons pressed, for example? How is the display read out? It makes no real difference whether the system interfaces are meant for people or other systems. They have to be addressed. These testability issues must be considered early in the development process to prevent unpleasant surprises later. It may be possible to add or enhance an interface in the design stage that will serve as a 'back door' for testing (and maintenance) purposes. An example is the serial port on mobiles. The software will require some modifications as well. Tools must be checked for suitability on the same points. It may be possible to adapt them or have them adapted to specific needs.

To test just the software, it can be compiled for a regular computer and run in a software simulation of its regular environment. This requires no hardware tools and usually makes addressing the interfaces much simpler. It is very useful for functional testing. The hardware does not even have to exist yet. For non-functional requirements such as performance and for the interfaces, integration testing with the real environment is still required.

Another issue to consider is whether any modifications to the system for testability or even observing system behaviour could have undesirable side effects. These could disturb the system and invalidate test results. Measuring a signal can weaken it and cause problems. Modifications can make software too large to fit in the often limited internal memory of embedded systems. The added logic on the target machine can then be limited to interfacing only and the rest performed on the host, thus minimizing memory footprint and processing time overhead on the target.

It is, of course, not necessary or even desirable to automate all tests. The benefit of automating a test depends on many factors such as its importance and how often it will be repeated during its lifetime. If the benefits of automating certain tests do not justify spending additional time or money, they should be performed manually. When test automation appears technically possible, some other issues require careful consideration.

Action words for effective testing

Very many test automation projects never fly or appear unsustainable. One important reason is that testers are often uncomfortable writing automated tests. Another is the amount of maintenance to test cases and test environment. Among the few serious general approaches for test automation, the 'action word' concept from the TestFrame method has been very successful. It has been applied in hundreds of projects, for GUI, web and dedicated systems (such as telecommunication equipment and digital TVs).

Action words are basically commands to the test environment, with or without arguments. The trick is in how they are defined. 'Test analysts' define product specific action words at the natural level of abstraction that they think about the tests, in their normal vocabulary. The action of storing a name and phone number in a mobile, for instance, requires pressing quite a few buttons. But as what individual buttons are pressed is irrelevant, they are better left implicit in an 'add number' action word. Details will not appear in test cases unless they need to be explicit. This includes test execution, tooling and interfacing details. Only the essence

remains. Figure 2 shows an example test case for a mobile phone, written in the usual spreadsheet format. This simple concept has some important consequences.

	code	
on	1234	
	name	number
add number	John Doe	0123456789
off		
	Code	
on	1234	
	Name	number
check number	John Doe	0123456789
	Name	
make call	John Doe	
	Name	
remove number	John Doe	
	Name	
check no number	John Doe	
off		

figure 2: Action words and their arguments: easy to read and low maintenance sensitivity

Automated test cases are now easy to read, write and maintain, even for programming illiterate testers. A test case can read almost like plain text, even for the test manager. It will also be concise and easy to review by other testers. Even error recovery can often be implicit. For developing all tests, the testers require only the action word definitions and the usual analytical skills, no programming skills or tooling details.

Test execution is taken care of by 'navigation software'. It implements the action words using the tools, providing the details that the test cases leave implicit. This software corresponds to the additional layer of test software in figure 1. It is best developed by software engineers, called 'navigation engineers'. They will need knowledge of tools and interfaces to translate the action words into actions at tool or system interfaces. They do not require the insight in the system of the testers.

The use of action words also strongly reduces maintenance sensitivity of tests and test environment. When changes in system functionality cause maintenance to test cases, the well-chosen action words (and navigation software) hardly change. Similarly, changes in interfaces or tooling will cause some maintenance to action word implementations, but if the system functionality does not change, neither do the test cases. Even switching to a different tool should have no impact on the test cases. The reduced maintenance sensitivity helps accommodate requirement changes during the development process. It also stimulates reuse of both tests and navigation software for other versions of the same product and related products.

These factors help avoid the common problems and achieve effective test automation. The above approach is useful for any system, be it GUI or dedicated. The test analysis process - developing test cases in action words - is also the same for all systems (although the nature of the tests will differ because the systems do). The main difference for dedicated systems is in implementing the action words.

For GUI systems this software is written in the scripting language of the tool that simulates user actions (keystrokes and mouse clicks). Navigation engineers are experts in GUI interfaces and their test tools. They will need to structure their code, but it is rarely complex.

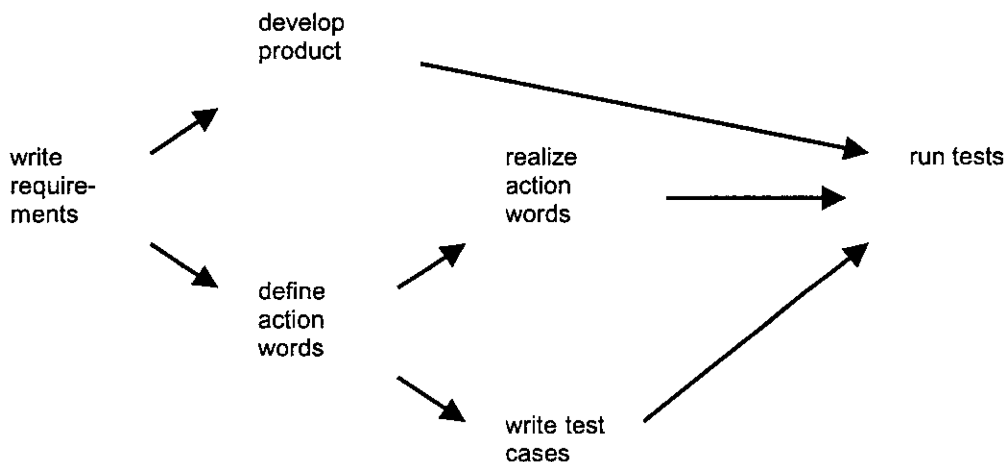


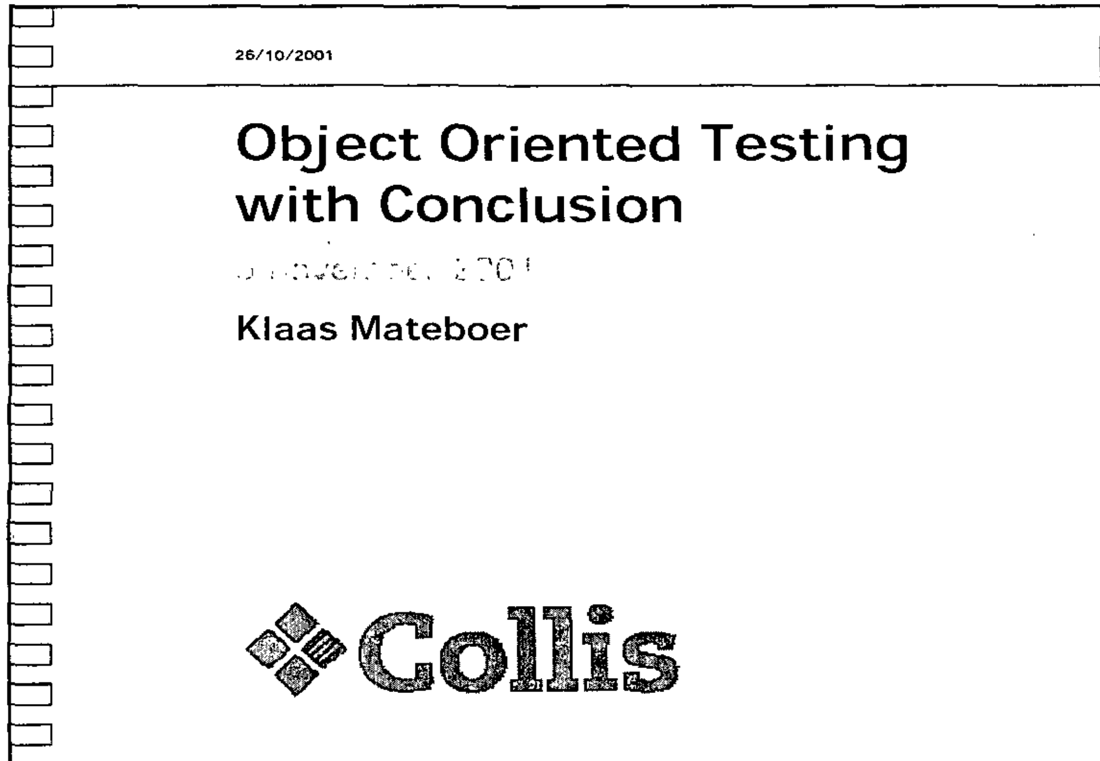
figure 3: Developing product, test cases and action words in parallel

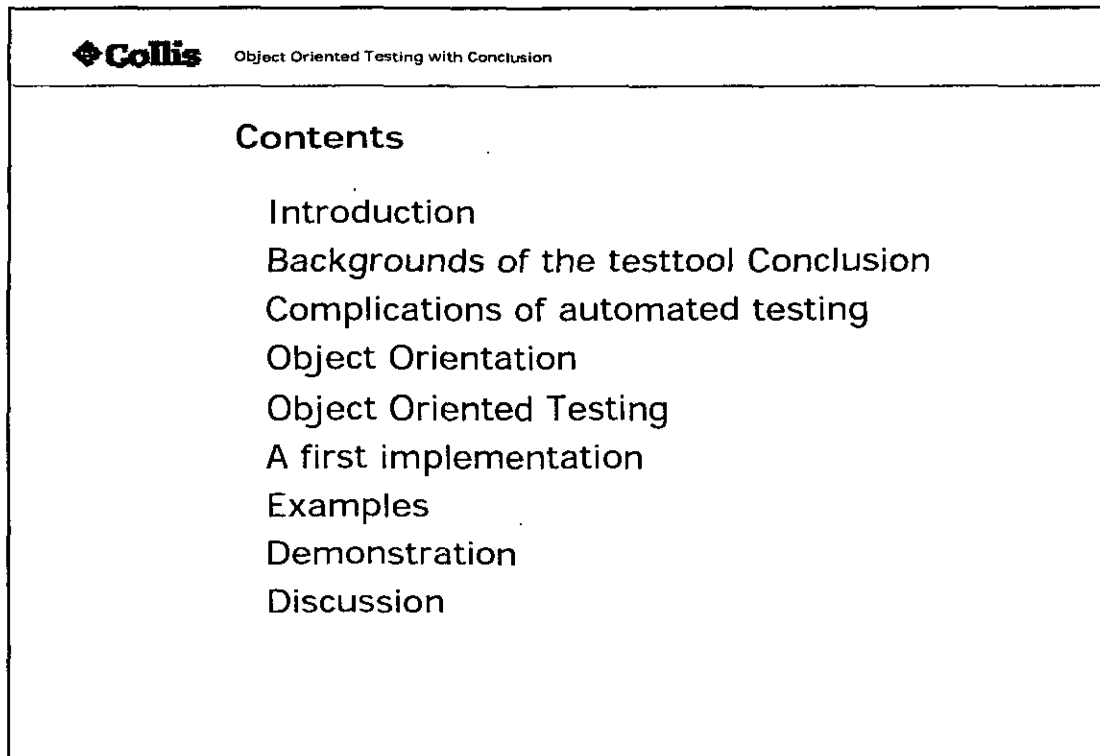
To test dedicated systems, the navigation software has to address other interfaces than the scripting language of a GUI tool. The exact interfaces vary greatly from system to system, but they often require writing special software. Sometimes calling operating system functions or existing library procedures suffices. Sometimes the navigation software is written in the scripting language of a tool, as for GUI systems. If some of the system's environment is

simulated, additional software is written around it. Sometimes complex data structures must be passed, such as the many kinds of messages from and to the mobile. All this makes the navigation software larger and more complex and therefore requires more software engineering skills to develop and maintain. Although the navigation software will be much less complex than the system itself, its development is best treated as a separate software engineering effort. It can take place in parallel with system and test development (see figure 3).

Doing it right

Because of the special interfaces, test automation for dedicated systems is different from GUI systems. Both the system and the tools must be (made) suitable. Addressing the system requires other knowledge and often more software engineering skills than for GUI systems. But as for other systems, the action word concept from the TestFrame method helps avoid some major pitfalls of test automation. It makes automated test cases easy to read and write. It clearly and strictly separates test development ('what') and test execution ('how'). It defines two roles and helps people in each role focus their creativity and skills. And it avoids the usual maintenance nightmare. Thus it helps create sustainable test automation solutions for both GUI and dedicated systems.


26/10/2001
Object Oriented Testing
with Conclusion
Klaas Mateboer
Collis


Collis
Object Oriented Testing with Conclusion
Contents
Introduction
Backgrounds of the testtool Conclusion
Complications of automated testing
Object Orientation
Object Oriented Testing
A first implementation
Examples
Demonstration
Discussion

Object Oriented Testing with Conclusion

Backgrounds of the testtool Conclusion

- SmartCard testtool Taste
  - Blackbox testing
  - Functional testing
  - Regression tests
  - Simple communication protocols
- Redesign and development since 1996
  - TTCN influences
  - ETDL script language combines basic programming facilities with communication skills, event handling, matching algorithms
  - PCO concept for clear and extensible interfacing with the system under test
  - FIF concept for flexible integration of foreign implemented functions (e.g. cryptography)

Object Oriented Testing with Conclusion

Application fields

- SmartCard testing
  - Simple communication protocols
  - Slave device
  - Cryptography
- SmartCard terminal testing
  - Master device
  - Multiple interfaces (cards, user, host)
  - More complicated communication protocols
- Host simulation and testing
  - Sophisticated communication and messaging
  - Multiprocessing
  - Database interfaces

Object Oriented Testing with Conclusion

Starting points

- All interfaces with the SUT are realised by PCO's
- Every interaction with the SUT is represented by a byte string
- Interface specific actions can be performed using control commands
- Test language includes basic facilities for operations on byte strings
- Complicated algorithms are available through FIF's

Object Oriented Testing with Conclusion

Complications

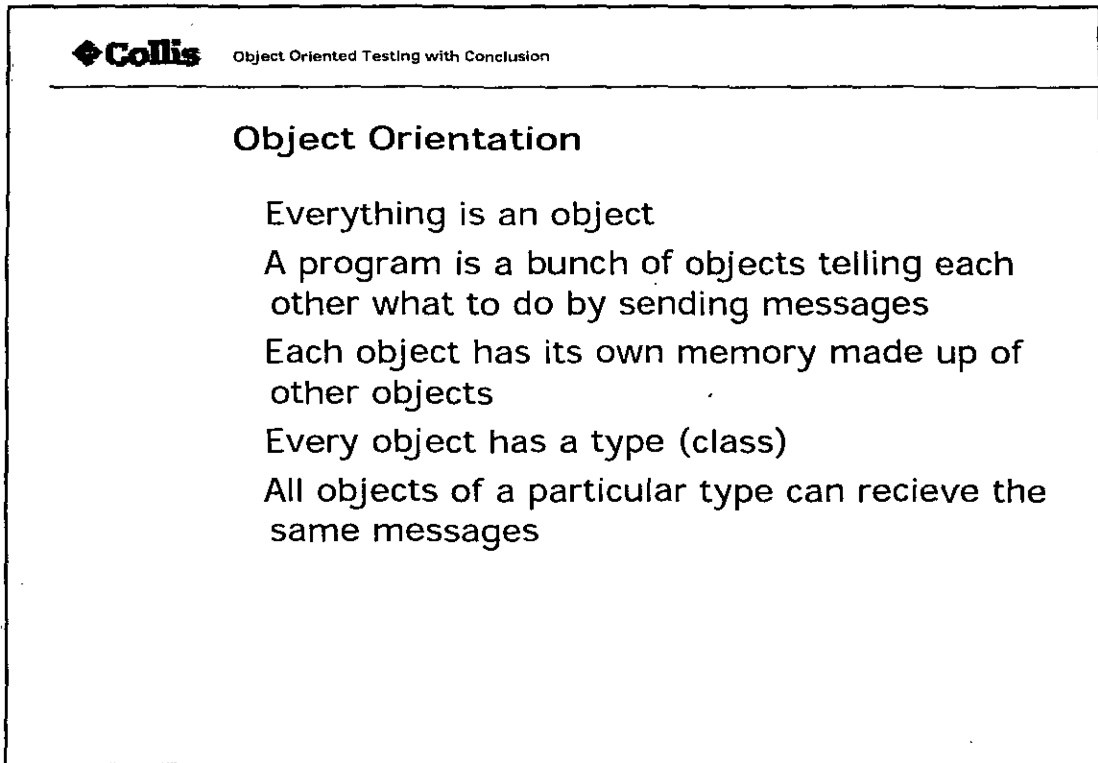
- Parsing complicated messages
- Functional tests at higher protocol layers
- Customisation of test reports
- Interaction with databases
- Component bus interfaces

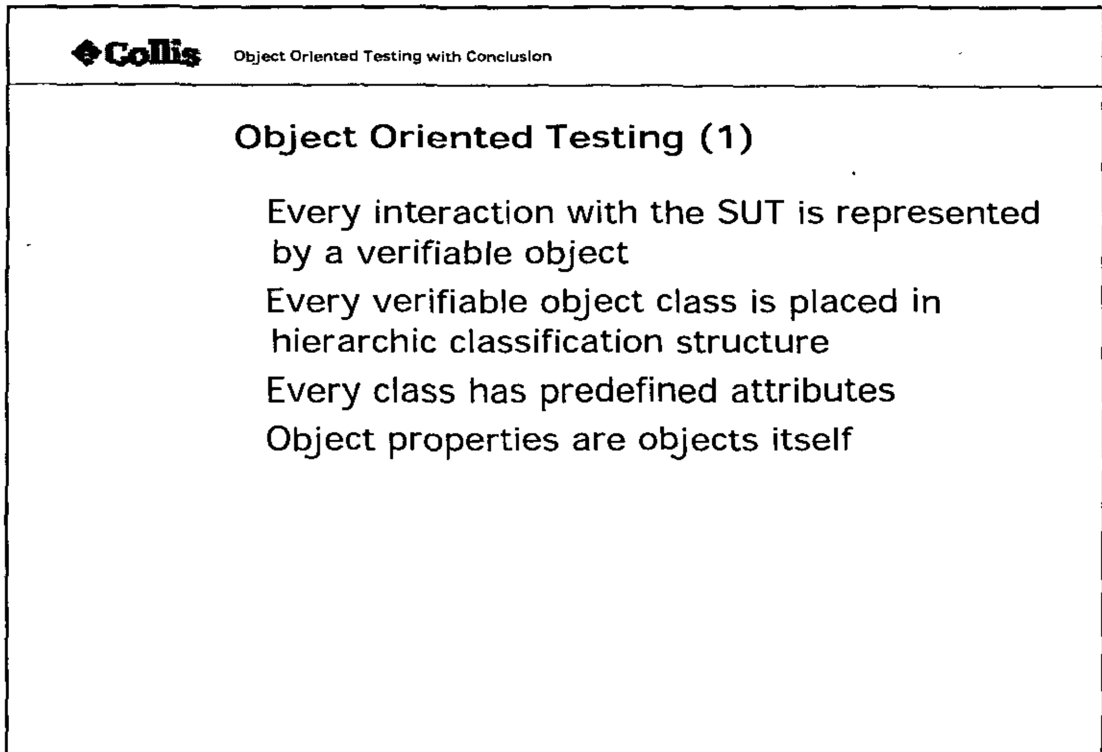
New goals

- Test description
  - Less imperative descriptions
  - Matching more than byte strings only
- Test environment
  - Protocol layering
- Report generation

What may be the solution?

- More intelligence in the test tool
  - Interpretation of events
  - Verification of messages
  - Representation of test results
- PCO's should be to the point
  - No protocol imlementations
  - No verification tasks
  - No reporting
- New constructions for matching
  - Different types of messages and events
  - Arrays and collections


Collis
Object Oriented Testing with Conclusion
Object Orientation
Everything is an object
A program is a bunch of objects telling each other what to do by sending messages
Each object has its own memory made up of other objects
Every object has a type (class)
All objects of a particular type can recieve the same messages


Collis
Object Oriented Testing with Conclusion
Object Oriented Testing (1)
Every interaction with the SUT is represented by a verifiable object
Every verifiable object class is placed in hierarchic classification structure
Every class has predefined attributes
Object properties are objects itself

Object Oriented Testing with Conclusion

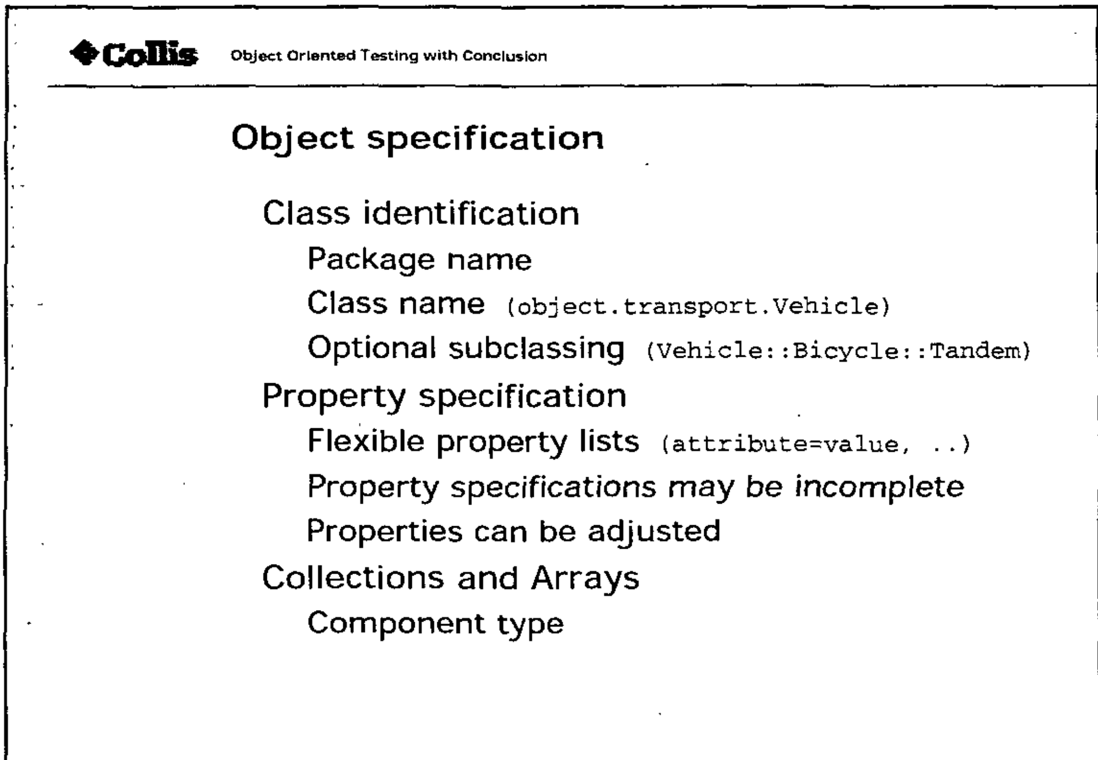
Object Oriented Testing (2)

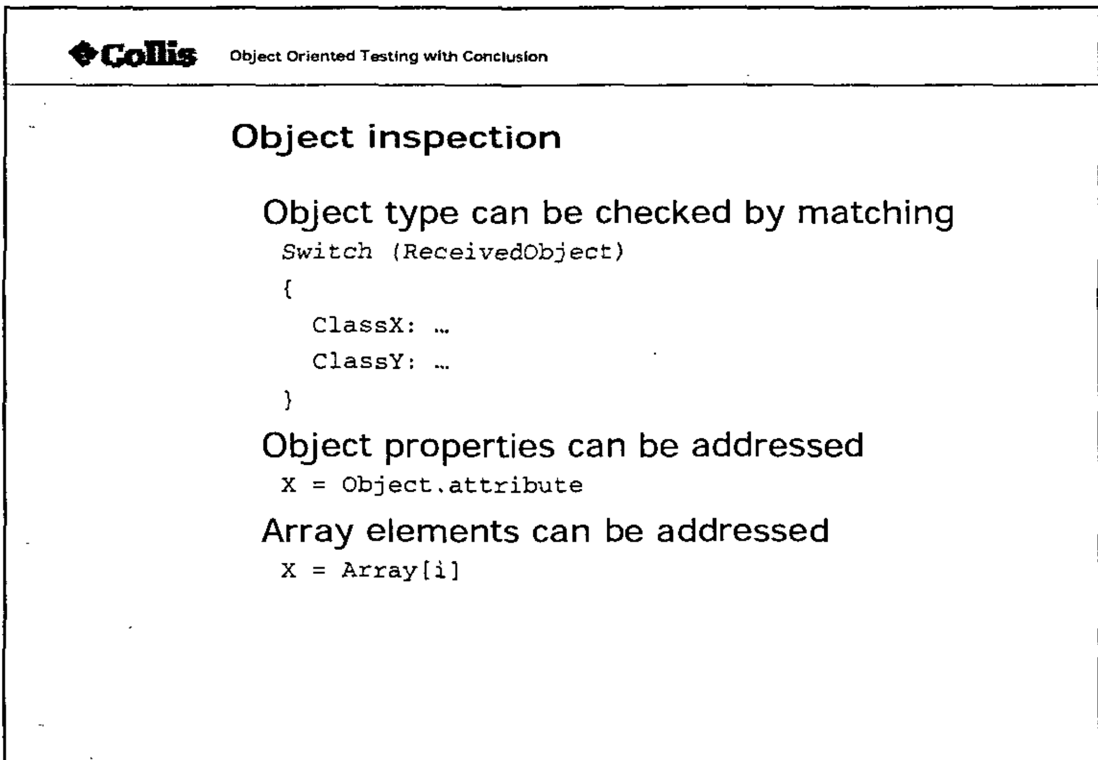
- The test language is extended with facilities for object (class/property) specification
- Matching based upon subclassing and optional property matching
- Verifiable objects handle translation of expressions and events
- Verifiable objects can handle part of their representation

Object Oriented Testing with Conclusion

Object classification

- Package structure for flexibility and maintenance
- Hierarchical class tree
- Inheritance of attributes
- Interfaces for standardised application areas


Collis
Object Oriented Testing with Conclusion
Object specification
Class identification
Package name
Class name (object.transport.Vehicle)
Optional subclassing (Vehicle::Bicycle::Tandem)
Property specification
Flexible property lists (attribute=value, ..)
Property specifications may be incomplete
Properties can be adjusted
Collections and Arrays
Component type


Collis
Object Oriented Testing with Conclusion
Object inspection
Object type can be checked by matching
Switch (ReceivedObject)
{
ClassX: …
ClassY: …
}
Object properties can be addressed
X = Object.attribute
Array elements can be addressed
X = Array[i]

Object manipulation

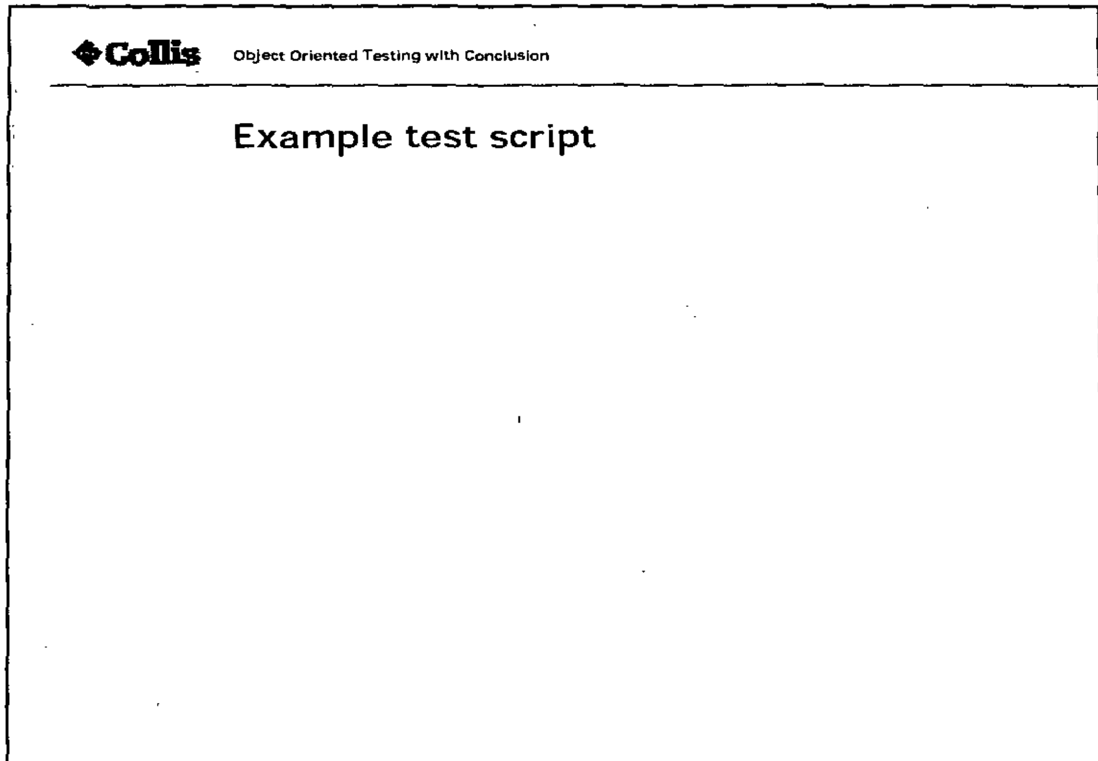
- Object type cannot change
- Object properties can be changed by respecification

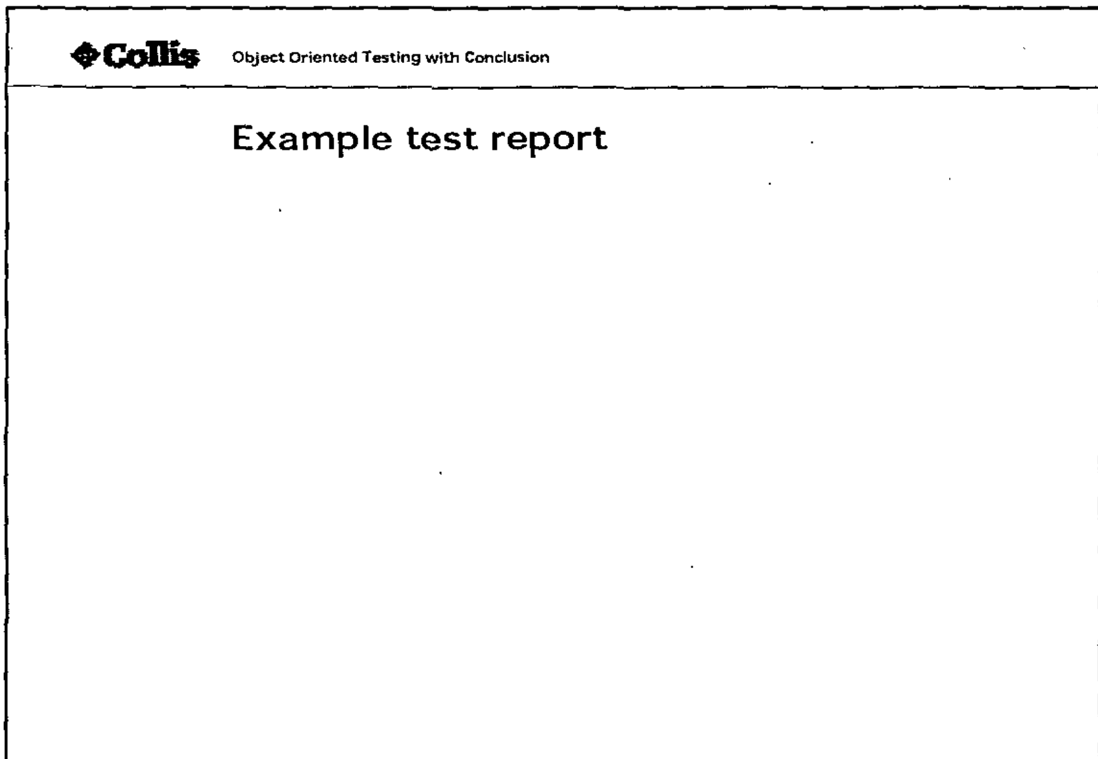
```
Object.attribute = newvalue
NewObject = OriginalObject(attribute=newvalue)
```

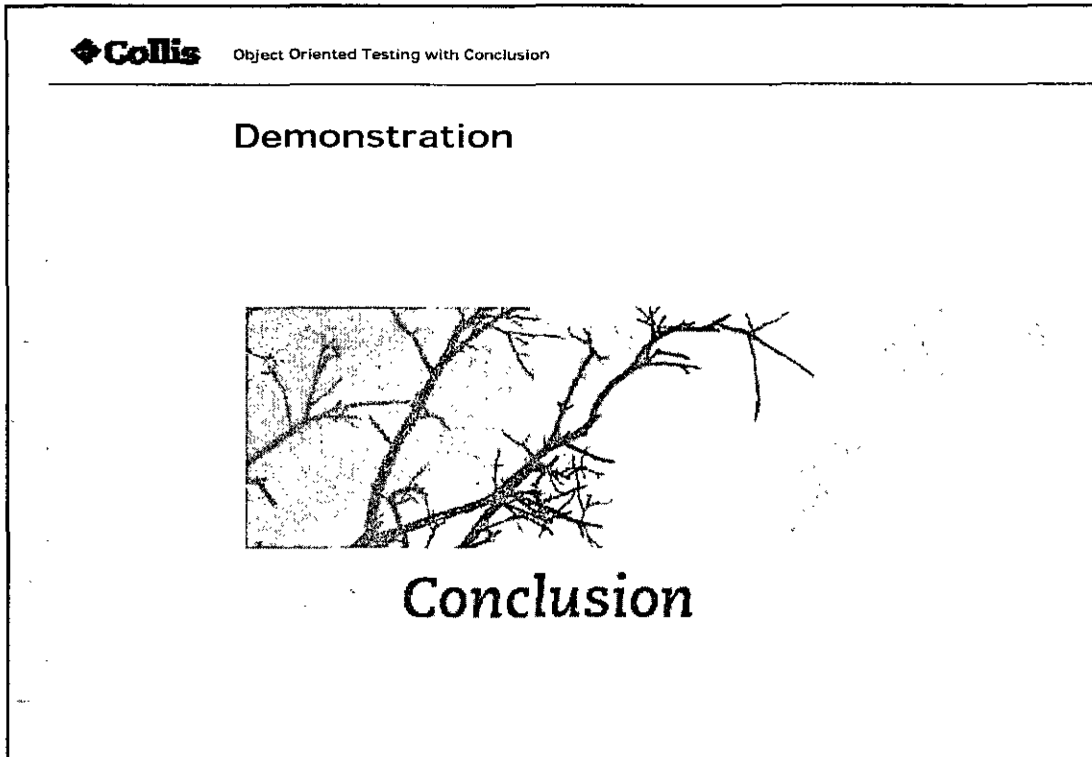
- Array elements can be reassigned

```
Array[i] = NewElement
NewArray = Array1 ++ Array2
NewArray = OriginalArray[i,j]
```

Example classification


Collis
Object Oriented Testing with Conclusion
Example test script


Collis
Object Oriented Testing with Conclusion
Example test report



Collis Object Oriented Testing with Conclusion

Discussion

Côte de Resyste

stw

Specification Based Testing:

Lessons from practical applications

René de Vries
(rdevries@cs.utwente.nl)

Lucent Technologies Bell Labs Innovations PHILIPS Let's make things better TU/e University of Twente The Netherlands

Côte de Resyste

University of Twente The Netherlands

Overview

- Scope and motivation
- Example: Highway Tolling System
- Evaluation and concluding remarks

Nederlandse Testdag 2001

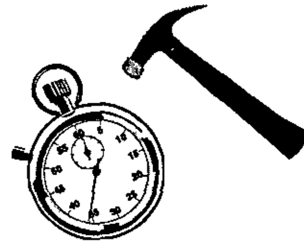
Côte de Resyste

University of Twente, The Netherlands

Testing

Testing:

- to check the quality of a system
- by performing experiments
- in a controlled environment



Software Testing:

- testing the quality of a software product

Primary Focus:

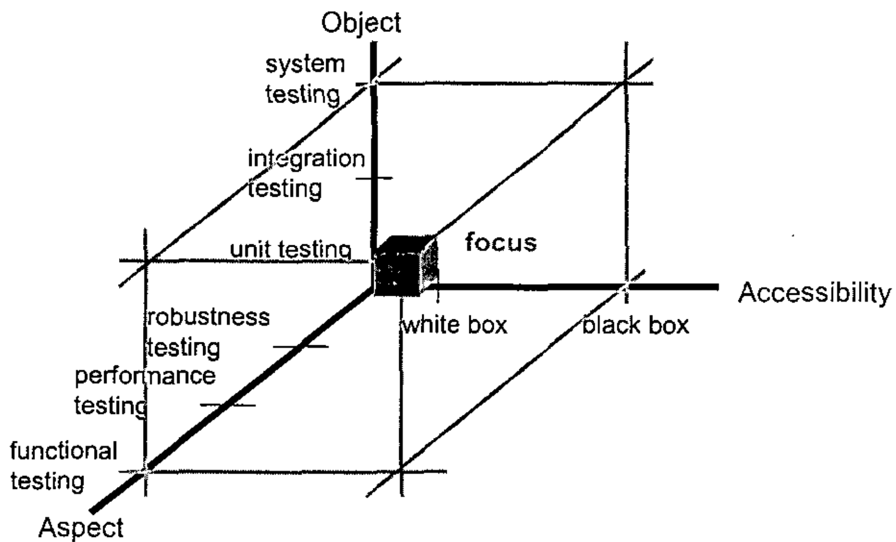
- *reactive* software systems

Nederlandse Testdag 2001

Côte de Resyste

University of Twente, The Netherlands

Types of testing



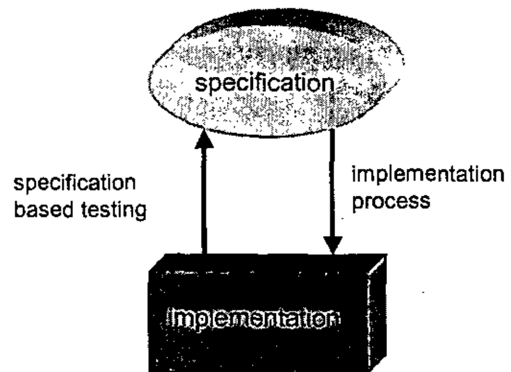
Nederlandse Testdag 2001

Côte de Resyste

Specification based testing

What?

- test functional behaviour
- of black box implementation
- with respect to specification



Why?

- to check correct functioning of implementation

Nederlandse Testdag 2001

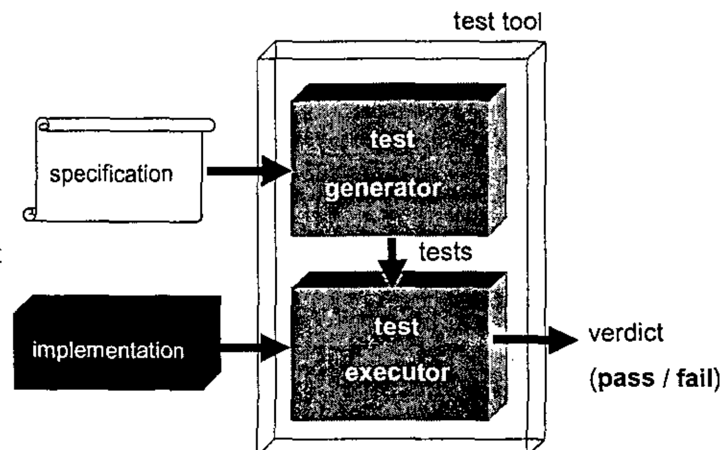
Côte de Resyste

Specification based testing: Ideal situation

1 generate test from specification

2 execute test against implementation

3 obtain verdict



Nederlandse Testdag 2001

Côte de Resyste

Problems of Testing

- What is correctness?
- Validity of the experiments
- Complexity of manual derivation
- Time consuming and laborious
- Inability of manual execution

Solution: Mathematics and Automation

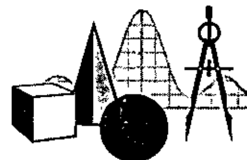
Côte de Resyste

Formal methods

- use mathematics to model relevant part of your system
- precise semantics: no room for misinterpretation
- allow formal validation and reasoning about systems
- amenable to tools: *automation*

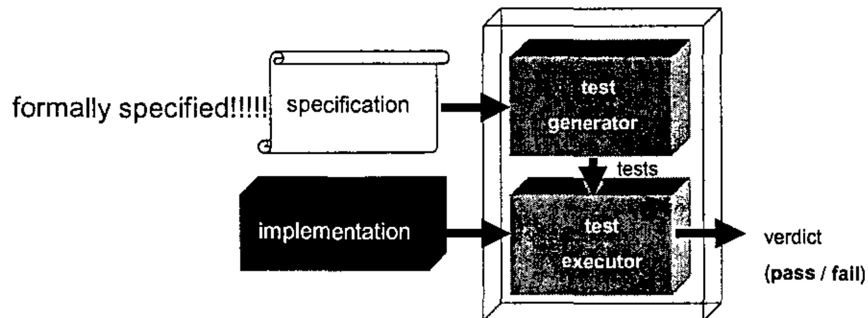
Examples

- Z, Temporal Logic, SDL, LOTOS, Promela, ...



Côte de Resyste

Formal methods and specification based testing



What?

- checking, by means of testing, whether an implementation is correct with respect to its specified behaviour
- where the specification is a formal description

Nederlandse Testdag 2001

Côte de Resyste

Côte de Resyste

(Conformance Testing of Reactive Systems)

- Research and development project
- Funding by: *Stichting voor Technische Wetenschappen (STW)*
- Effort: 15fte (STW) + 8 fte (partners) for 4 years (Januari 1998 - December 2001)
- Partners: industry (Philips and Lucent Technologies) and academia (TUE, UT)

Aim:

Develop methods and techniques and build tools for automated specification based testing based on *formal theory* and validate these in practice.

Nederlandse Testdag 2001

Côte de Resyste

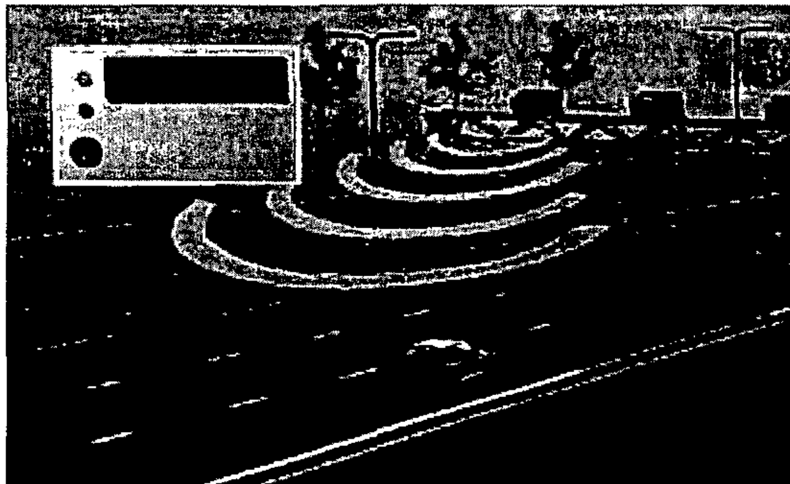
Case Studies

- Academic:
  - conference protocol (reference)
- Industrial:
  - Easylink (Philips)
  - TV components (Philips)
  - V5EES (Lucent Technologies)
  - CBC (CMG)
  - Rekeningrijden (Interpay)

Nederlandse Testdag 2001

Côte de Resyste

Example: Rekeningrijden



Nederlandse Testdag 2001

Côte de Resyste

Characteristics

- Simple protocol
- Parallism
- Real-time requirements
- Encryption

Nederlandse Testdag 2001

Côte de Resyste

Phases for Automated Testing

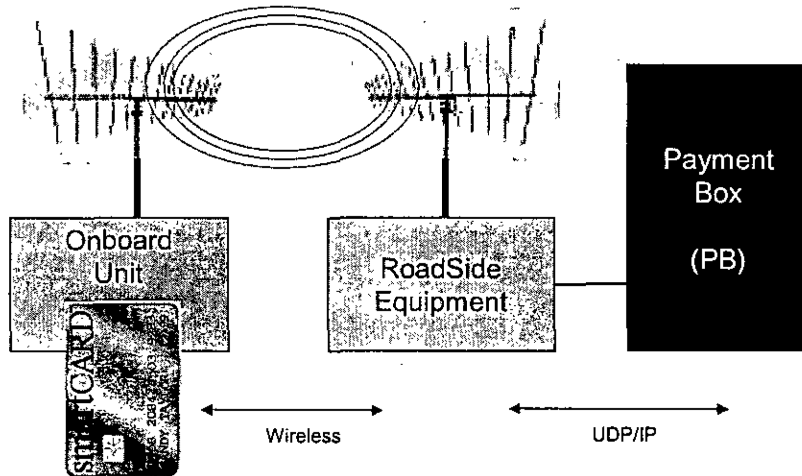
- IUT study
  - informal and formal specification
- Tools
  - semantics and openness
- Test environment
  - test architecture, implementation, SUT specification and testing
- Test execution
  - campaigns and execution

Nederlandse Testdag 2001

Côte de Resyste

University of Twente
The Netherlands

Highway Tolling System

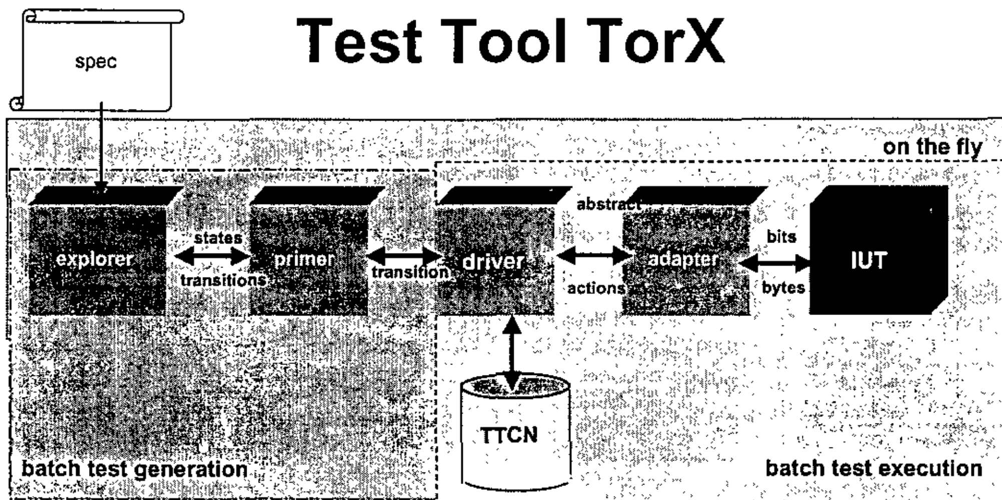


Nederlandse Testdag 2001

Côte de Resyste

University of Twente
The Netherlands

Test Tool TorX

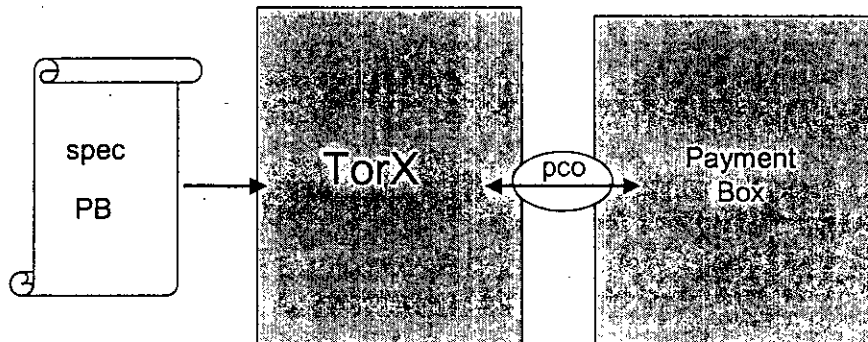


focus on on-the-fly testing

Nederlandse Testdag 2001

Côte de Resyste

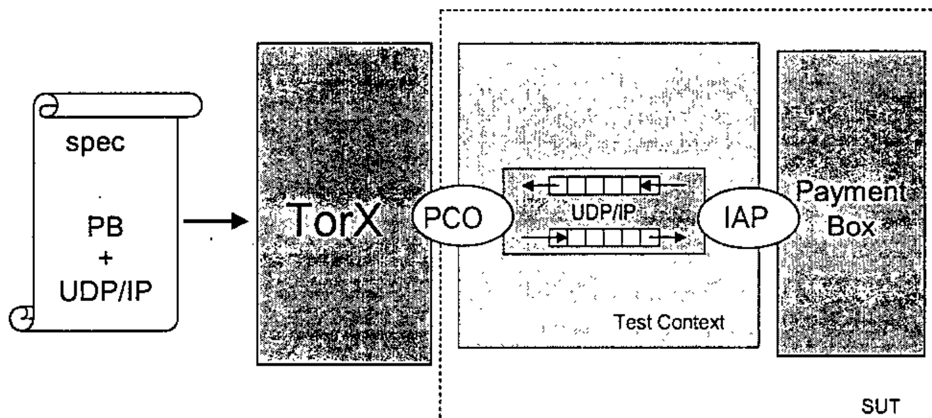
Test architecture I



Nederlandse Testdag 2001

Côte de Resyste

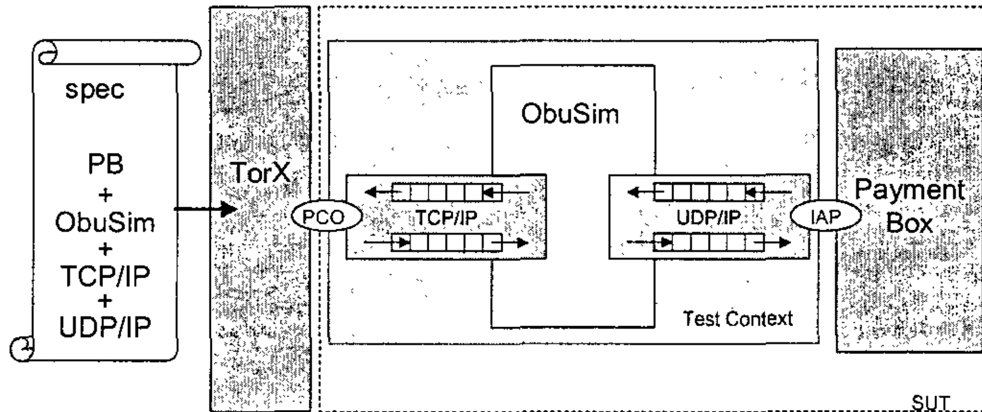
Test architecture II



Nederlandse Testdag 2001

Côte de Resyste

Test architecture III



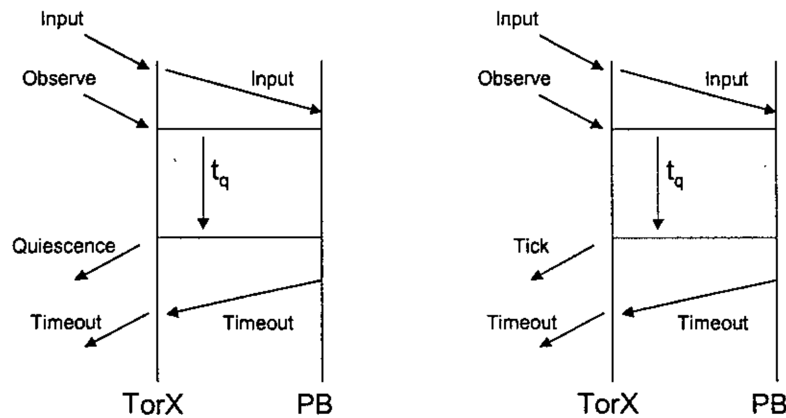
Côte de Resyste

Test Execution

- Campaigns
  - Management of test tool configurations
  - Management of IUT configurations
  - Steering the test derivation
  - Archiving results
- Execution results
  - 2 errors: 1 validation and 1 during testing

Côte de Resyste

Problem: Quiescence in ioco



Spec := Spec + Tick

Nederlandse Testdag 2001

Côte de Resyste

Major Lessons

- $S_{test} \neq S_{validation}$
- Engineering of the test environment is laborious
- Automated testing is very flexible
- Test campaigns support bookkeeping and control on experiments
- How to cope with real time requirements
  - efficient computation for on-the-fly testing
  - lack of theory

Nederlandse Testdag 2001

Côte de Resyste

University of Twente
The Netherlands

Concluding remarks

- Automated testing is:
  - beneficial (high volume and reliability)
  - flexible (adaptation and many configurations)
- More needed:
  - Adapter implementation
  - Real Time theory
  - Campaigns and result analysis

Step ahead in testing realistic industrial applications

Nederlandse Testdag 2001

Côte de Resyste

University of Twente
The Netherlands

More information:
• http://fmt.cs.utwente.nl/cdr

Thanks:
CdR partners and Interpay B.V.

Nederlandse Testdag 2001

Côte de Resyste

University of Twente
The Netherlands

Questions

Nederlandse Testdag 2001

Prisoner's Dilemma in Software Testing

Loe Feijs
Eindhoven University of Technology

Abstract

In this article the problem of software testing is modeled as a formal strategic game. It is found that for certain values of the productivity and reward parameters the game is essentially equivalent to the Prisoner's Dilemma. This means that the game has a unique Nash equilibrium, which is not optimal for both players, however. Two formal games are described and analyzed in detail, both capturing certain (though not all) aspects of real software testing procedures. Some of the literature on the Prisoner's Dilemma is reviewed and the results are translated to the context of software testing.

Keywords: software testing, software quality, game theory, Nash equilibrium, prisoner's dilemma.

1 Introduction

Software is playing an increasingly important role in modern society. Not only in office software and computer games, but also in embedded systems such as in televisions, telephony exchanges, cars, etc. computer programs of considerable size are essential. Software bugs constitute a serious problem [1]. Software testing is frequently used to find errors so they can be repaired and hopefully, the software quality improved. In view of the societal relevance of software quality, it makes sense to consider software quality as an economic issue and to use rational methods when studying it. In this article we use concepts from game theory for that purpose. The problem of software testing is modeled as a formal strategic game. One of the findings of this article is that this game is of a special nature, known as Prisoner's Dilemma [2].

The article is structured as follows. Sect. 2 introduces the necessary game theory, which includes the Prisoner's Dilemma. Sect. 3 describes the role of testing in software engineering. Sect. 4 describes an idealized testing game and Sect. 5 discusses a subtle variation of the same game. We have an example of a concrete testing game, providing support for and adding plausibility to the game of Sect. 4. Its presentation is very technical, however; therefore all the details are described in App. A. In App. B this concrete testing game is analyzed. In Sect. 6 the effect of multiple performance levels is investigated. In Sect. 7 several known results on the Prisoner's Dilemma are summarized and translated to the context of software testing. Finally Sect. 8 contains some concluding remarks.

2 Prisoner's Dilemma

There are two types of games that are studied in game theory: zero sum games and non-zero sum games. Zero sum games obey the rule that the sum of the payoffs to all players equals zero. An example is the game of chess where a player receives one point if he wins, -1 point if he looses and 0 points in case of a draw. There are two players and three outcomes: white wins and black looses, white looses and black wins, and a draw (no one wins). The payoffs are $\langle 1, -1\rangle$, $\langle -1, 1\rangle$ and $\langle 0, 0\rangle$, respectively. In each case, the sum equals zero, e.g. if white wins the payoffs are given as the pair $\langle 1, -1\rangle$ so the sum $1 + (-1) = 0$. Zero sum games were studied by Zermelo and Von Neumann [3] who established solution concepts and proved their

existence using minimax analysis and fixed point theories. Zero sum games model situations where players have a conflict of interests such that their interests are completely opposite.

Non-zero sum games, by contrast, model situations in which there is a conflict of interest, but where the opposition is not complete. To a certain extent, the players can have shared interests, but their interests are not completely aligned either. Throughout this article the class of non-zero sum games is needed.

Another distinction usually made in game theory is between strategic games and extensive games. In a strategic game all players choose their actions once in a simultaneous fashion whereas an extensive game is more general; in an extensive game it is possible to perform several actions in a sequential fashion (as for example in chess and checkers). Throughout this paper we focus on strategic games, which are formally introduced next.

A strategic game is a triple $\langle N, (A_i), (u_i)\rangle$ where N is a finite set of players and for each player $i \in N$ there is a set A_i of so-called actions. For each player $i \in N$ there is a function $u_i : A \rightarrow \mathbb{R}$ that assigns a payoff to each tuple of actions. Here $A = \Pi_{i \in N} A_i$, or in the special case that the size of N equals two, $A = A_1 \times A_2$. The payoffs give rise to a preference ordering $\geq_i$ on A, one ordering for each player, defined by $a \geq_i b$ iff $u_i(a) \geq u_i(b)$. The absolute values of the payoffs are considered irrelevant in the sense that only the induced preference ordering counts. Throughout this article the number of players equals two. Thus N contains two elements and without loss of generality it can be assumed that $N = \{1,2\}$. A pair of actions $\langle a, b\rangle$ for $a \in A_1$ and $b \in A_2$ is called an action profile.

A simple example of a strategic game is BoS, the Battle of the Sexes, in [5] explained as Bach or Stravinsky. It is here used as an example to introduce a convenient tabular notation called payoff matrix. The actions of player 1 correspond to the rows of the matrix. The actions of player 2 correspond to the columns of the matrix. Each entry in the matrix contains a pair: the payoff of player one, followed by the payoff of player 2. The payoff matrix of BoS is:

	B	S
B	2,1	0,0
S	0,0	1,2

The action sets are $A_1 = A_2 = \{B, S\}$, that is, each player has to indicate his choice for a concert of music, going to either Bach (B) or Stravinsky (S). Their main concern is to go together, but one person prefers Bach and the other prefers Stravinsky.

The interesting question is which action profiles are to be chosen by rational players. A very important concept for studying this question is the Nash equilibrium (the original publication is [4]; we follow the presentation of [5]). A Nash equilibrium (N.E.) of a strategic game $\langle N, (A_i), (u_i)\rangle$ for $N = \{1,2\}$ is an action profile $\langle a^*, b^*\rangle$ with the property that $\langle a^*, b^*\rangle \geq_1 \langle a, b^*\rangle$ for all $a \in A_1$ and similarly $\langle a^*, b^*\rangle \geq_2 \langle a^*, b\rangle$ for all $b \in A_2$. The intuition is that a Nash equilibrium is a consistent expectation pair in the sense that player 1, if he is rational, sticks to his choice a^* which is better for him than any other choice (assuming player 2 does not deviate from his b^*) and conversely for player 2. The concept of Nash equilibrium is illustrated by the game of BoS, which has two Nash equilibria, viz. $\langle B, B\rangle$ and $\langle S, S\rangle$. The preferences are shown in the diagram of Figure 1 which is like the payoff matrix except for the arrows which indicate how player 1 can improve a certain action profile by another action profile (according to the vertical arrows) and similarly for player 2 (according to the horizontal arrows).

After these preparations the Prisoner's Dilemma is introduced. The Prisoner's Dilemma is the two-player strategic game with the following payoff-matrix:

	C	D
C	$-1,-1$	$-3,0$
D	$0,-3$	$-2,-2$

The game was proposed by Merril Flood and Melvin Dresher in 1950 in a slightly different setting. The name 'Prisoner's Dilemma' was proposed by Albert Tucker, who found the anecdote of the two prisoners. In [2] the game is explained as follows:

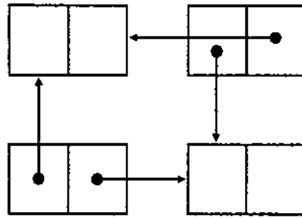


Figure 1: Improvements for the BoS game.

> Two members of a criminal gang are arrested and emprisoned. Each prisoner is in solitary confinement with no means of speaking to or exchanging messages with the other. The police admit they don't have enough evidence to convict the pair on the principal charge. They plan to sentence both to a year of prison on a lesser charge. Simultaneously, the police offer each prisoner a Faustian bargain. If he testifies against his partner, he will go free while his partner will get three years in prison on the main charge. Oh, yes, there is a catch ... If *both* prisoners testify against each other, both will be sentenced to two years in jail.

This explanation motivates the sets $A_1 = A_2 = \{C, D\}$ where D means 'Defect' and C means 'Cooperate'. Since the absolute values are irrelevant, the Prisoner's Dilemma can equally well be described by the following payoff matrix (taken from [5]):

	C	D
C	3,3	0,4
D	4,0	1,1

There is something paradoxical about the Prisoner's Dilemma which is the fact that the only N.E., the action profile $\langle D, D\rangle$ with payoff pair $\langle 1, 1\rangle$ is inferior to the pair $\langle C, C\rangle$ with payoff pair $\langle 3, 3\rangle$. It is inferior for both players. The improvements diagram of Figure 2 demonstrates that $\langle D, D\rangle$ is the only N.E.:

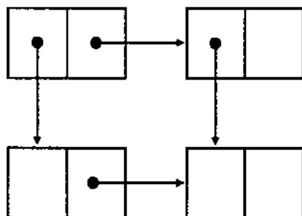


Figure 2: Improvements for the Prisoner's Dilemma.

3 Software testing

Since the invention of stored-program computers in the 1940s the importance of computer programs has been steadily increasing. Today, many millions of people rely on software such as Unix, Windows, MS Office for important aspects of their work. Televisions, telephony exchanges, cars, airplanes, mobile phones, etc. each contain thousands, sometimes millions of lines of embedded code. Experience shows that it is very hard to guarantee the correctness of these programs (often referred to as software). Sometimes the problems seem out of control, as illustrated by the problems reported in [1]: Software's Chronic Crisis. Gibbs mentions the

example of the baggage-handling system of Denver's new international airport that, for nine months, was held captive by "Lilliputians-errors in the control software". Shaw of Carnegy Mellon, Bourgonjon of Philips and others note how software is exploding. Gibbs writes about bugs, small glitches, the density of errors and so on. Software testing is used in serious attempts to improve the situation by detecting errors as soon as possible. As Hall puts it in [1]: "The benefits of finding errors at that early stage is enormous". Testing plays an important role in the present industrial practice, particularly when the software systems are very large. As an example we mention the development of Windows 2000, being 29 million lines of code. We quote from [6] where it is reported that: "[ ] at the level of size and complexity of Windows 2000, writing code was no longer the central activity. Indeed, testing and debugging have accounted for between 90 and 95 percent of the work". It is generally believed that large-scale software projects have to spend 50% or more of their effort in testing.

There is a large literature on the art and science of software testing. Formal method specialists stress the fact that testing can reveal errors but can not prove the absence of errors (Dijkstra EWD340); both research and case studies are done to explore the power of correctness proofs and automated analysis tools (model checkers). Despite these research efforts, most practitioners consider some form of testing indispensible. In the field of protocol conformance testing, the topic of testing is subject of much research and development. We mention the language TTCN [7] and Finite State Machine based methods such as those mentioned in [8], [9] and Labeled Transition System methods, for example [10].

Most authors in formal testing research adopt the viewpoint that there should be a formal specification which is used as a starting point for procedures to generate tests, execute them, and determine their coverage. The main goal seems to be that the efficiency, or error detection power of the tests-to-be-derived is optimized.

Useful as this may be, little or no attention is given to the fact that there is an opposition of interests between the tester and the implementor. The goal of the tester is to find errors, whereas it is easily observed in practice that an implementer likes his program to be shown correct (as he often believes it is). The practical advise to separate these roles, letting the tester be another person than the implementer is well-known for this reason.

In this article we do not just seek an optimizing procedure for the tester, but we focus on the opposition of interests. Since game theory has a vocabulary and analysis methods for situations of opposite interests, these can be applied to software testing. This is undertaken in the next section.

4 An idealized testing game

In this section a very simple and yet fairly general model is proposed first. The model is nothing but a payoff matrix for two players: an implementer (player 1) and a tester (player 2). After that the model will be refined to bring in the probabilistic aspects that ocur when neither the implementation nor the test are perfect (so both PASS and FAIL are possible outcomes, each with its own probability). The model is abstract in the sense that it does not tell what an implementation or a test looks like. A more concrete example model, showing that the abstract model makes sense, is postponed until App. A.

The Idealized Testing Game (ITG) is the two-player strategic game $\langle N, (A_i), (u_i) \rangle$ where $N = \{1, 2\}$ and $A_1 = A_2 = \{p, q\}$, with the intuition that p means 'poor' (bad) and q means 'quality' (good). The u_i are given by the payoff matrix:

	p	q
p	2,2	0,3
q	3,0	1,1

The motivation for this payoff matrix is as follows. The implementer is rewarded if he delivers an error-free, or almost error-free piece of software. The tester is rewarded if he performs a thorough testing job. In practice it is hard to tell whether the software is error-free or not;

that is precisely what testing is meant for. Therefore in ITG the implementer is rewarded if no error is found (an outcome denoted as PASS). The tester should be eager to find errors. Therefore in ITG, the tester is rewarded if at least one error is found (an outcomed denoted as FAIL). The implementer chooses between doing a poor job, delivering software containing errors, and doing a quality job, delivering error-free or almost error-free software. The tester chooses between doing a poor job, performing a test with low error detection capability, and performing a quality test, which takes more effort, but which is more likely to reveal errors. The payoffs for the action profile $\langle p, p \rangle$ are 2 for the implementer and 2 for the tester. Both do a poor job, so yes, the software contains errors, but at the same time there is a low error detection capability by the tester's choice. Some of the errors are found, some are not. Both the implementer and the tester are equally well rewarded; this explains both payoffs that are 2 (recall that the absolute value is irrelevant). Another situation occurs when an almost perfect implementation is investigated in an almost perfect test. But now the payoffs are 1 since both players have to make serious investments in effort (time), and perhaps also in costs for more sophisticated personal education, tools etc. The costs of these efforts, here estimated to have a value of 1, are to be subtracted from the rewards, both for the implementer and the tester.

Next consider the action profile $\langle q, p \rangle$, that is, an almost perfect piece of software investigated by a poor test. Clearly this gives a PASS. The implementer's reward increases from an average of 2 to a certain value of 4. But the effort is higher too, so the implementer gets 3. The tester has no costs for the effort, but no rewards either. The effect of reward outperforms the effort (=1) so the tester's payoff is 2 less than his payoff for $\langle p, p \rangle$; the tester gets 0. Conversely, $\langle p, q \rangle$ gives a FAIL with payoffs 0 and 3.

The payoff matrix of ITG is the payoff matrix of a Prisoner's Dilemma. There is one N.E., the action profile $\langle q, q \rangle$. In the N.E. both players choose to do a quality job.

Next this model is to be refined in order to bring in the probabilistic aspects. The refined model is called ITG′. It is defined by a 4×4 payoff matrix. From ITG′ the earlier model ITG can be derived in a formal way: the four choices for the implementer are reduced to two choices, each of which is a lottery over two alternative implementations of equal quality level (and similarly for the other player). By this approach, the numbers from the ITG payoff matrix can be retrieved later (in other words: ITG′ will provide a more detailed motivation for ITG).

The refined Idealized Testing Game (ITG′) is the two-player strategic game $\langle N, (A'_i), (u'_i) \rangle$ where $N = \{1, 2\}$ as before and where $A'_1 = A'_2 = \{p1, p2, q1, q2\}$. The u'_i are given by the payoff matrix:

	p1	p2	q1	q2
p1	4,0	0,4	0,3	0,3
p2	0,4	4,0	0,3	0,3
q1	3,0	3,0	3,−1	−1,3
q2	3,0	3,0	−1, 3	3,−1

The motivation for this payoff matrix is as follows. The implementer has always several alternative design choices, even after he has made a conscious or unconscious decision to spend the effort corresponding to a poor job or the effort corresponding to a quality job. In real life this design space is huge but here it is assumed to be a two-element set. It is the set $\{p1, p2\}$ for 'poor' and the set $\{q1, q2\}$ for 'quality'. The reward for a programmer doing a good job, as witnessed by a PASS, equals 4 (think of 4\$, 4K\$ or 400K\$ depending on the size and complexity of the specification). In the case of a FAIL the programmer receives reward 0. The tester gets 4 for a FAIL, 0 otherwise. A poor implementation and a poor test cost 0. A quality implementation costs 1 (the effort), which is counted negatively. The same holds for the test cost. It is assumed that $\langle p1, p1 \rangle$ and $\langle p2, p2 \rangle$ produce a PASS whereas $\langle p1, p2 \rangle$ and $\langle p2, p1 \rangle$ produce a FAIL. But if the programmer chooses *any* of his quality alternatives $q1$ or $q2$ he enforces a PASS whenever the tester chooses one of the poor alternatives. And so on, as conveniently summarized by the following matrix (call this a verdict matrix):

	p1	p2	q1	q2
p1	PASS	FAIL	FAIL	FAIL
p2	FAIL	PASS	FAIL	FAIL
q1	PASS	PASS	PASS	FAIL
q2	PASS	PASS	FAIL	PASS

This concludes the motivation of ITG′.

This game ITG′ has no N.E. For example, consider the action profile $\langle p1, p1\rangle$ having payoffs 4,0. Then the tester can improve by choosing $p2$ instead of $p1$. The resulting action profile $\langle p1, p2\rangle$ has payoffs 0,4. Then the implementer can improve by choosing $p2$. The action profile $\langle p2, p2\rangle$ yields 4,0 so the tester improves to $\langle p2, p1\rangle$. The action profile $\langle p2, p1\rangle$ yields 0,4 so the implementer improves to $\langle p1, p1\rangle$. The 'improvements' go round in circles. Of course the reason is that the implementer has no reason to distinguish whether $p1$ or $p2$ is better because he has no way to know what the tester will do ($p1$ or $p2$).

Therefore the following two-step strategy is reasonable: first choose between p and q, then conduct a lottery over the alternative implementations. In reality an implementer does not feel like throwing dice concerning his implementation decisions; he tries his best with the selected time frame, but yet he makes mistakes at places where he is unaware of it and that is modeled as a random device.

In order to calculate the average payoffs, the payoffs are multiplied by a weighting factor. The sub-matrix where the implementer takes one of the p choices and the tester takes one of his p choices, contains four entries with equal probabilities. These must be 25%. The weighting matrix is:

	p1	p2	q1	q2
p1	25%	25%	25%	25%
p2	25%	25%	25%	25%
q1	25%	25%	25%	25%
q2	25%	25%	25%	25%

Performing an element-wise multiplication of the ITG′ payoff matrix and the weighting matrix we get:

	p1	p2	q1	q2
p1	1,0	0,1	0,0.75	0,0.75
p2	0,1	1,0	0,0.75	0,0.75
q1	0.75,0	0.75,0	0.75,−0.25	−0.25,0.75
q2	0.75,0	0.75,0	−0.25,0.75	0.75,−0.25

Taking this latter matrix, the four weighted payoffs can be added for each of the four sub-matrices. For the $\langle p, p\rangle$ sub-matrix we must (player-wise) add 1,0 and 0,1 and 0,1 and 1,0 which means that the average payoffs for $\langle p, p\rangle$ are 2,2. The result is:

	p	q
p	2,2	0,3
q	3,0	1,1

This is easily recognized as ITG. So ITG comes out as an abstraction of ITG′.

Although ITG and ITG′ are based on reasonable assumptions they are very abstract. The syntax and semantics of the programs and the details of the testing procedures are not elaborated and there is not even a difference between an implementer and a tester. We have an example of a concrete testing game TCTG (Text Copy Testing Game), providing support for and adding plausibility to ITG and ITG′. In TCTG the task for the implementer is to transcribe a given text, perhaps from one character set to another, or perhaps even simpler, to type precisely the characters of a given string. The length of the specification and the length of the

implementation are the same, L say. The task of the tester is to select one or more positions in the range 1 to L. If the implementation and the specification differ at one or more of the selected positions then an error is found and the verdict is FAIL. Otherwise the verdict is PASS. The presentation of the Text Copy Testing Game is very technical, however. Therefore all the details are described in App. A. In App. B this concrete testing game is analyzed.

Discussion: ITG is based on ITG′ which contains reasonable assumptions about the test process such as the assumption that a poor-poor confrontation and a quality-quality confrontation yield a 50%/50% ratio of PASS and FAIL. Another assumption is that the reward or success is 4 times the effort needed to perform a quality job. This factor of 4 is a parameter of the game (call it the reward/effort ratio, symbol R). If the ratio $R = 4$ then ITG results, essentially a Prisoner's Dilemma. Taking other values, other games result, some of which have another nature than the Prisoner's Dilemma. For arbitrary ratio R the adapted payoff matrix of ITG is:

	p	q
p	$\frac{1}{2}R,\frac{1}{2}R$	$0,R-1$
q	$R-1,0$	$\frac{1}{2}R-1,\frac{1}{2}R-1$

Its is interesting to consider values of R that are below 4; the nature of the game changes at $R = 2$. For example, taking $R = 1\frac{1}{2}$ the payoff matrix turns into:

	p	q
p	0.75,0.75	0,0.5
q	0.5,0	−0.25,−0.25

This is not a Prisoner's Dilemma. There is one N.E., viz. $\langle p,p \rangle$ and the corresponding payoffs 0.75,0.75 are better than the payoffs for any other action profile (for both players). If this is interpreted in the usual way by assuming that both player and tester behave rational and hence take this N.E., then it is a natural explanation to say that there is not enough incentive for both players to choose the quality alternative: the expectation of the reward does not outperform the effort.

5 What if the implementer moves first?

There is an assumption underlying the ITG that can be questioned, viz. that the implementer and the tester choose their performance level simultaneously. This is motivated by the way many software engineering projects are organised. As soon as the requirements analysis phase has been completed and the software specification is available, not only the implementer starts working, but also the tester. A large part of the tester's work consists of choosing test cases and carefully describing them. This work should not wait until the implementer is ready because of the usual requirement to keep the overall project duration as short as possible (another part of the work is test execution, which has to wait for the implementation anyhow). For the main development of the present paper we adopt the viewpoint that testing is a strategic game which means that both players move simultaneously, as motivated by the parallelism in the development project. It is also motivated by the fact that even after the implementer has delivered his implementation, the tester does not know the implementer's performance level (usually, testing is needed for that).

But, as a short side-line we shall briefly analyse a sequential version of ITG, wich we call ITG_{seq}. In ITG_{seq} the payoff matrix is:

	p	q
p	2,2	0,3
q	3,0	1,1

which is the same as for ITG. The difference is that in ITG_{seq} the implementer chooses first. Then, knowing the implementer's choice, the tester chooses. The motivation for this could be that in certain specific situations the tester can easily sample the implementer's work in order to estimate the performance level (for example if the code contains many deeply nested IF THEN ELSE statements, bugs are likely).

Next ITG_{seq} is analysed according to a max-maximization procedure (this is a two-phase optimization similar to the well-known min-max procedure for zero-sum games). If the implementer chooses p then the tester is left with the upper sub-matrix:

	p	q
p	2,2	0,3

The assumedly rational tester maximizes his own payoff and therefore chooses q. In this case the implementer receives payoff 0. Alternatively, if the implementer chooses q then the tester is left with the lower sub-matrix:

	p	q
q	3,0	1,1

The tester, maximizing his own payoff, chooses q. In this case the implementer receives payoff 1. The implementer is facing a choice between p with payoff 0 and q with payoff 1. Assuming the implementer is rational, he must choose q. Then the tester chooses q. So the solution of the max-maximization analysis is the (sequential) action profile $\langle q, q \rangle$. As it happens, this is the same as the N.E. of the ITG.

6 Adding an extra performance level

One of the objections one could have against the relevance of the ITG and the TCTG of App. A as a model of an implementer's behavior and a tester's behavior is the binary choice with respect to the performance level. Perhaps a real implementer does not want to choose between two extreme values of 'poor' and 'quality' levels, or between $Q = 20\%$ and $Q = 50\%$. A real implementer sees a whole range of performance levels, so perhaps he chooses between three levels (or even more). It is not a priori clear what implications this has for the nature of the game. ITG and TCTG make it difficult to choose between the poor level and the quality level (because of the paradoxical nature of the Prisoner's Dilemma) but will the dilemma disappear if there is a third, intermediate level? In order to investigate this, we construct another game called ITG3 which is a 3×3 strategic two-player game.

The game ITG3 is obtained from ITG by introducing an intermediate performance level halfway between 'poor' and 'quality'. The same notational devices used in App. B are used here. In order to simplify the calculations we approximate the effort values by linear interpolation (the rewards are always the same, but the effort part is varying). Therefore the refined payoff matrix is:

	p	pq	q
p	4,0 /0,4	4,−0.5 /0,3.5	4,−1/0,3
pq	3.5,0/−.5,4	3.5,−0.5/−0.5,3.5	3.5,−1/−0.5,3
q	3,0 /−1,4	3,−0.5 /−1,3.5	3,−1/−1,3

The weighting matrix is also taken to be the result of linear interpolation between the idealized values of P(FAIL) which are 0%, 50% and 100% in ITG. Therefore the weighting matrix is:

	p	pq	q
p	0.5 /0.5	0.25 / 0.75	0/1
pq	0.75/0.25	0.5 / 0.5	0.25/0.75
q	1/0	0.75 / 0.25	0.5 /0.5

The payoff matrix and the weighting matrix can be multiplied in an element-wise fashion to get:

	p	pq	q
p	2,0 /0,2	1,−0.125/0,2.625	0,0/0,3
pq	2.625,0/0.125,1	1.75,−0.25/−0.25,1.75	0.875,−0.25/−0.375,2.25
q	3,0/0,0	2.25,−0.375/−0.25,0.875	1.5,−0.5/−0.5,1.5

And by pair-wise adding the payoffs for PASS and FAIL the following payoff matrix is obtained (call this abstract 3×3 game ITG3):

	p	pq	q
p	2,2	1,2.5	0,3
pq	2.5,1	1.5,1.5	0.5,2
q	3,0	2,0.5	1,1

This is again a game with one N.E. at $\langle q, q \rangle$ but as in the Prisoner's Dilemma the payoffs 1,1 for the N.E. are less than certain other payoffs, such as the 2,2 for $\langle p, p \rangle$. The improvements diagram for ITG3 is shown in Fig. 3. From this analysis we see that introducing an intermediate

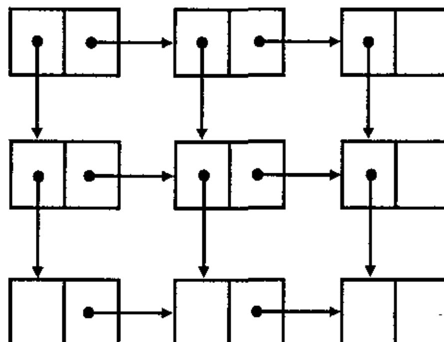


Figure 3: Improvements for the ITG3 game.

performance level does not change the essential nature of the game.

7 Testing is a Prisoners Dilemma; so what?

Consider a software development laboratory in which there are programmers (=implementers) and testers. The efforts and rewards are determined according to the principles such as those of ITG and TCTG. Programmers receive a fixed basis salary, but if they choose to do a 'quality' job, they have to spend more hours (which could have been spent otherwise in a profitable way, which means that the programmer pays for extra efforts). Moreover the programmer receives a bonus (the reward) if the subsequent test results in a PASS verdict. A similar payment system is adopted for the tester. There are other conceivable set-ups as well, for example introducing a programmer's boss who chooses the programmer's performance level and pays him accordingly; in this case it is the boss who plays the game, but it does not make much difference.

Now from the viewpoint of the software lab's manager: is it good or bad that there is a Prisoner's Dilemma going on in his lab? There are several answers possible, depending on the type of theoretical results that are employed and depending on the level of belief that these theoretical results apply to real life. Six different types of theoretical results are discussed:

- The concept of N.E. is a straightforward theoretical concept of 'solution' for a two player strategic game. If it is assumed that rational players choose the N.E. and if both the

programmer and the tester are rational, then they must choose the 'quality' performance levels; for the lab manager this means that they do their best to deliver quality software, which is the manager's (and the customer's) best outcome. It is best for the manager, not for the players. Looking at this way, ITG is good.

- Already in the early days of the development of game theory, serious doubt has been raised whether real people would behave rational in the sense of choosing the N.E., even in paradoxical situations. The Prisoner's Dilemma was conceived precisely as a tool for investigating this. In the Flood-Dresher experiment [2] done in 1950 a sequence of Prisoner's Dilemma games was conducted by the same pair of players; mutual cooperation was the most common outcome: 60 of the 100 games. Mutual defection, the N.E., occurred only 14 times. Translating this to the ITG, mutual cooperation means $\langle p,p \rangle$ and mutual defection means $\langle q,q \rangle$. If the programmer and the tester behave as in this experiment, their behavior is bad from the lab manager's (and the customer's) point of view.

- If the same game is played repeatedly by the same pair of players, it is not correct from a game-theoretical view to consider these as independent games (as immediately remarked by John Nash in his reaction to the Flood-Dresher experiment, see [2]). The sequence is to be considered as one game in which the two players move pairwise, using the knowledge about the other player's strategy (as built-up so far) in each successive move. In game theory this is known as an 'extensive game with perfect information and simultaneous moves' (see [5] 6.3.2). Now each player has to decide upon a strategy which contains his initial move and also all the answers to all sequences of moves of the other player. Special examples of such strategies are finite state machines. In [5] Sect. 8.4 several such strategy machines are defined. For example one machine M_1 for player 1 works as follows: play C (cooperate) as long as player 2 plays C; play D (defect) for three periods, and then revert back to C, if player 2 plays D when he should play C. So the other player is being "punished" for three periods for playing D and then "forgiven". The machine M_2 of player 2 starts by playing C and continues to do so if the other player chooses D. If the other player chooses C then it switches to D, which it continues to play until the other player again chooses D, when it reverts to playing C. If two players play Prisoner's Dilemma using these finite state machine strategies, the game goes into cycling behavior with a cycle-length of 5. Both $\langle C,C \rangle$ and $\langle D,D \rangle$ occur in the cycle, next to $\langle C,D \rangle$ and $\langle D,C \rangle$. If the programmer and the tester work according to such strategies (and other strategies such as the 'grim' strategy in [5] or the well-known 'tit-for-tat' strategy) then the lab manager is likely to observe cycles; it can be expected that the cycles are irregular because of the random effect of the $\langle p,q \rangle$ and the $\langle q,p \rangle$ action profiles (recall that ITG is based on average payoffs).

- It is possible to consider the higher-level strategic game in which each player has to choose a multi-move strategy once; for example the programmer could choose to always take move D and the tester could behave as defined by machine M_2. The interesting question now is whether there exist some kind of optimal multi-move strategy (some clever machine, perhaps) that is optimal for player 1 and similar for player 2. In [5] Sect. 8.10.3 the following result is given: "if the strategic game G has a unique N.E. payoff profile then for any value of T the action profile chosen after any history in any subgame perfect equilibrium of the T-period repeated game of G is a N.E. of G." Here 'subgame perfect equilibrium' refers to a special type of N.E. for extensive games, taking certain credibility considerations into account (see [5] Sect. 6.2 for the details). If the programmer and the tester have to play ITG in a finite repetition and if they behave rational then this result means that they choose $\langle q,q \rangle$, which is good from the lab manager's viewpoint. However there is evidence that experimental subjects do not behave in a way that is consistent with this result (see the discussion in [5] Sect. 8.2, and also see the Flood-Dresher experiment [2]).

- When considering infinitely repeated games there is the complication that it is not a-priori

clear how the payoffs of infinite sequences are to be defined (just adding them leads to infinite values). There are several alternative definitions: the discounting criterion, the limit of means criterion and the overtaking criterion. The set of equilibria is huge. In [5] Sect. 8.2 Osborne remarks: "the fact that it [the behavior of experimental subjects] is consistent with some subgame perfect equilibrium of the infinitely repeated game is uninteresting since the range of outcomes that are so-consistent is vast." Moreover software developments do not run ad infinitum without bringing in new people or changing the rules.

- It is possible to consider players which choose multi-move strategies (grim, tit-for-tat, etc.) but not just two fixed players; rather an evolving population of players is considered. In [11] three different arrangements are discussed. The first arrangement is a historic sequence of tournaments organized by Axelrod in the later 1970. Colleagues (game theorists) submitted strategies. The simple strategy tit-for-tat won (start with cooperative response and then always repeat the other player's previous move). The second arrangement is a computer simulation of large populations of certain basic types of players equipped with stochastic strategies. Many mutation-selection rounds are performed. As a result, the average payoff in the population can change suddenly. Most of the time, either almost all members of the population cooperate, or almost all defect. The longer the system was allowed to evolve, the greater likelihood "for a cooperative regime to blossom." In heterogeneous populations tit-for-tat is not always superior; it is outperformed by another strategy ('Pavlov', see [12] and [13]). The third arrangement is a population that lives on a two-dimensional grid. Players only interact with eight immediate neighbors. A lone cooperator will be exploited by the surrounding defectors and succumbs. But four cooperators in a block can hold their own. Geometric patterns arise that wander accross the board. In one example population [11] the percentage of cooperators gradually takes a stable value of about 32%. Translating these assumptions to the world of software development we can think of a high-tech area (or perhaps the whole world) with software companies, providing programming services and testing services to a limited set of neighboring companies. For details see [11] and [14].

8 Concluding remarks

The results of our investigations show that if the reward for passing tests for the programmer is high enough and the reward for finding an error for the tester is high enough, there is essentially a Prisoner's Dilemma game hidden in the interaction of a programmer and a tester. The main cause of this seems to be in the assumption that the development laboratory manager cannot measure the quality of the software directly; he only sees PASS or FAIL and therefore he cannot distinguish a bad or lazy programmer in combination with a bad or lazy tester from a good and eager programmer and an a good eager tester (in both cases there is a mixture of PASS and FAIL verdicts on average).

The simplest theoretical solution of the game, the N.E. seems to suggest that mutual defection is the expected outcome. The Prisoner's Dilemma is often seen as "an interesting metaphor for the fundamental biological problem of how cooperative behavior may evolve and be maintained" [14]. In the context of testing the situation is reversed since a testing process is set-up with a deliberate, even desirable, conflict of interest built into it. So the N.E. means that both players choose 'quality' rather that 'poor', which is good from the viewpoint of the software development laboratory and its customers. Looking at sequences of games, the theory does not have much predictive power. Experiments and computer simulations tend to show complex, oscillatory behaviors, mostly showing cooperation. Two-dimensional evolutionary simulations show complex moving patterns.

The complex, oscillating and moving patterns are quite consistent with the image of a software crisis and hard-to-eliminate bugs sketched in [1]. In view of our findings it seems wise for a software development lab manager to make sure that he separates the roles of programmers

and testers and that he makes sure there is enough reward for a programmer to achieve PASSes and for a tester to achieve FAILs. Although our game-theoretical analysis was done using numerical values for the payoff matrix, it is certainly possible that rewards in real life include items such as a feeling of professional responsibility and personal pride.

Disclaimers: it is necessary to relativize the findings because there are certain assumptions behind the models that need not hold in real life situations. First of all, this is the assumption that there is a precise specification given to both players that guarantees that the verdicts PASS and FAIL are not subject of dispute. Although a frequent assumption in research on formal test generation, this need not be true in real development projects. Another assumption is that testing by a professional tester is the only way of determining the quality of the programmer's work. This assumption does not hold if there are other mechanisms at work such as code-walkthroughs, beta-testing, or program correctness proving. Yet another assumption is that neither defect-free software nor full-coverage tests are possible (at or near $Q = 100\%$ and $P = 100\%$ the theory turns into a degenerate case). Finally the ITG and TCTG do not cover the intricacies of real software development: writing a program involves subtle tradeoffs between development time, run-time efficiency, code-size, reusability etc. We constructed another game where finite state machines were tested by finite sequences (programmer effort being automaton size, test effort being test sequence length). We found a Prisoner's Dilemma in this too (details not included in the present article). But, similarly to the text copy testing game this is still an enormous simplification. Programming languages such as C++ and Java are very complex and so are test languages such as TTCN; moreover defining and measuring metrics for programmer's productivity, software quality and test coverage are rich research subjects by themselves.

Acknowledgements

Some of the ideas behind this work described in this article originated in the context of the Côtes de Resyste project and during discussions with Nicu Goga, Sjouke Mauw and Jan Tretmans. The author likes to thank Pieter Cuijpers for his constructive feedback on earlier versions of the paper.

A The text copy testing game

In order to validate the concepts of ITG we develop a more concrete game in which real testing is going on. This game, called TCTG, for Text Copy Testing Game is still not really about software testing, but at least it is about testing. It will be formally analyzed to see whether it is approximated by ITG.

In TCTG The task for the implementer is to transcribe a given text, perhaps from one character set to another, or perhaps even simpler, to type precisely the characters of a given string. The length of the specification and the length of the implementation are the same, L say. The task of the tester is to select one or more positions in the range 1 to L. If the implementation and the specification differ at one or more of the selected positions then an error is found and the verdict is FAIL. Otherwise the verdict is PASS. If the tester chooses several positions, they all have to be different.

As in ITG the implementer and the tester are rational players who choose a performance level first and after that perform a job according to the expectations set by the chosen performance level.

It is convenient to introduce numerical values for the performance levels. In ITG there are two levels but here many levels could exist. The performance level for the implementer is modeled as a variable Q (the Quality of the implementation). The performance level for the tester is modeled as a variable P (the Power of error detection of the test). Both Q and P are in the range from 0 to 1 (the values of 0 and 1 are included). Sometimes they are conveniently expressed as percentages. So instead of choosing a p (poor) performance level the implementer may choose $Q = 25\%$ and instead of a q (quality) performance level the implementer may choose $Q = 50\%$. In the same way the tester may choose $P = 25\%$ or $P = 50\%$, respectively. As in ITG the idea is that equal performance levels lead to a probability of error detection P(FAIL) of about 50%. Indeed, it will turn out that $Q = P = 50\% \Rightarrow$ P(FAIL) $= 50\%$ but because of some of the details of the text copying and testing procedures this idea only holds by approximation for other value pairs for Q and P. The performance level variables are a useful concept but for each value pair of Q and P, the precise probability P(FAIL) has to be calculated as a function $f(Q, P)$. It is natural to demand that the efforts of the implementer and the tester increase monotonically when Q and P increase, respectively. It would be nice for the implementation effort to be a linear function of Q and for the testing effort to be a linear function of P. In the text copy testing game this turns out to hold for P indeed but for Q it only holds as a very rough approximation. This concludes the general introduction of the numerical performance levels; next they must be defined for the specific situation of text copying and text testing.

It is an error if at a specific position the typed character differs from the specified character. In a text of L characters any number of errors E between 0 and L is possible (0 and L inclusive). The implementer's performance level Q is defined by $Q = \frac{1}{E+1}$. For example when $L = 100$, the range of Q is from 0.99% (100 errors) to 100% (no errors). Other definitions are conceivable, for example $\frac{L-E}{L}$ but we consider it inappropriate to assign a relatively high value of 50% to a text in which half of the characters are wrong (in software code it would be ridiculous to even look at software in which 50% of the code statements are wrong). The tester's performance level P is defined by $P = \frac{T}{L}$ where T is the number of positions tested. The range of P is from 0% (nothing tested) to to 100% (everything tested). The advantage of the current definitions is that $Q = 50\%$ and $P = 50\%$ nicely outbalance each other.

Next the relations between the efforts and the performance levels must be defined. These relations depend on the way of working of the implementer and the tester. For the implementer it is assumed that there is a mechanism such that spending more effort leads to a reduction in the number of errors. An example of such a mechanism is a *voting text editor*: each character is typed n times for a given number $n \in \{1, 3, 5, 7, \dots\}$ etc. and whenever at least $\frac{n+1}{2}$ of these typings are the same, that determines a winning character which goes into the implementation. Otherwise a special error characters is taken. Note that this relation satisfies the monotonicity condition Although the usage of voting mechanisms is not widespread in software engineering, the idea of using redundancy in software design has been proposed under the name "distinct

programming" by Tom Gilb [15].

The implementer has a built-in error rate e, for example $e = 0.1$ which cannot be changed (except by exploiting the voting mechanism). For the tester it is assumed that his effort is proportional to the number of positions tested.

Given these assumptions, the numerical values of Q and P can be determined for various effort values. For $L = 100$ and $e = 0.1$ the expected number of errors with a repetition rate of 1, denoted as $\mathcal{E}(\#\text{ errors} \mid 1\times)$ equals $e \times L = 10$. So $Q = \frac{1}{11} = 9\%$. For triple repetition $\mathcal{E}(\#\text{ errors} \mid 3\times) = (\text{ P(2 errors)} + \text{P(3 errors)}) \times L = (3.(1-e).e^2 + e^3) \times L = (3 \times 0.9 \times 0.01 + 0.001) \times L = 0.028 \times L = 2.8$ whence $Q = \frac{1}{2.8+1} = 26\%$. For five-fold repetition $\mathcal{E}(\#\text{ errors} \mid 5\times) = (\frac{5\times4}{2} \times (0.9)^2 \times 0.001 + 5 \times 0.9 \times 0.0001 + 0.00001) \times L = 0.86$ whence $Q = \frac{1}{1.86} = 54\%$. Also $\mathcal{E}(\#\text{ errors} \mid 7\times) = (\frac{7\times6\times5}{3\times2} \times (0.9)^3 \times 0.0001 + \frac{7\times6}{2} \times (0.9)^2 \times 0.00001 + 7 \times 0.9 \times 0.000001 + 0.0000001) \times L = 0.27$ whence $Q = \frac{1}{1.27} = 79\%$. Finally $\mathcal{E}(\#\text{ errors} \mid 9\times) = 0.08$ so $Q = \frac{1}{1.08} = 93\%$. The following table summarizes the above findings (for $L = 100$ and $e = 0.1$):

#repetitions	Q
1	9%
3	26%
5	54%
7	79%
9	93%

The relation between the effort (here the number of tested positions) and the power of error detection P (the tester's performance level) is summarized by the following table (for $L = 100$):

#positions	P
10	10%
30	30%
50	50%
70	70%
90	90%

For a game theoretic analysis only the reward/effort ratio is important and therefore the fixation of the absolute effort values is postponed.

B Analysis of the text copy testing game

The first question is how P(FAIL) depends on Q and P as a function $f(Q, P)$ to be determined. First consider a few simple cases for $L = 100$. Let there be one error ($Q = 50\%$) and let there be 50 tested positions ($P = 50\%$). Then P(FAIL) $= 50\%$ (the probability that this error is one of these 50 positions out of 100). Let there be 3 errors, $E = 3$, so $Q = \frac{1}{E+1} = \frac{1}{4}$ and let there be 50 tested positions ($P = 50\%$ again). P(FAIL) $\approx 1 -$ (P(given error not found))^3 $= 1 - (1-P)^3 = 1 - (0.5)^3 = 0.875$, that is 87.5%. In general, if $E \ll L$ then P(FAIL) $\approx 1 - (1-P)^E$ and since $E = \frac{1}{Q} - 1$ the following formula for $f(Q, P)$ is adopted:

$$f(Q, P) = 1 - (1-P)^{\frac{1}{Q}-1}$$

For which combinations of Q and P does P(FAIL) $= 50\%$ hold? Easy calculations show $f(0.5, 0.5) = f(0.25, 0.21) = f(0.125, 0.094) = 50\%$. This shows that the iso-P(FAIL) line of 50% does not always run through the points defined by the equation $Q = P$, but for the points shown here it is pretty close.

Next it is time for playing the game. The implementer chooses between $Q = 20\%$ and $Q = 50\%$. After that he chooses randomly among $\binom{100}{4}$ implementations (if $Q = 20\%$) or 100 implementations (if $Q = 50\%$). The tester chooses between $P = 20\%$ and $P = 50\%$. After

that he chooses randomly among $\binom{100}{20}$ tests (if $P = 20\%$) or $\binom{100}{50}$ tests (if $P = 50\%$). The cost of effort difference between $Q = 20\%$ and $Q = 50\%$ is put equal to 1 (think of it as 1\$, perhaps). Interpolating between 1 and 3 repetitions, $Q = 20\%$ occurs at 2.3 repetitions and interpolating between 3 and 5 repetitions, $Q = 50\%$ occurs at 4.7 repetitions; in other words, the $4.7 - 2.3 = 2.4$ extra repetitions (on average), being $2.4 \times L = 240$ key strokes cost 1\$ extra, or 0.42 \$c per key stroke. Similarly the cost of the difference between $P = 20\%$ and $P = 50\%$ is put equal to 1. The difference is 30 positions so each position selected and inspected by the tester costs 3.33\$c. It is convenient to assume for $Q = 20\%$ and $P = 20\%$ the efforts to be equal to 0 (this is convenient; it is irrelevant for the game-theoretic analysis). Let the reward/effort ratio (R) be set to 5.

What is P(FAIL)? It depends on Q and P. For $Q = P = 20\%$, P(FAIL) $= 0.59$. For $Q = 50\%$, $P = 20\%$ it is found that P(FAIL) $= f(50\%, 20\%) = 1 - (0.8)^{\frac{1}{0.5}-1} = 0.2$ and as calculated before, for $Q = P = 50\%$, P(FAIL) $= 50\%$. So there is a difference with the ITG, where $\langle q, p \rangle$ turned P(FAIL) into 0, whereas here the analogous $\langle Q = 50\%, P = 20\% \rangle$ still leaves a non-neglectable P(FAIL) $= 0.2$ and similarly $\langle p, q \rangle$ gives P(FAIL) $= 1$ in ITG but the analogous probability is only $= 1 - (0.5)^{\frac{1}{0.2}-1} = 0.9375$ here. It will be interesting to see whether it still is a Prisoner's Dilemma.

On the basis of the abovementioned assumption the payoff matrix is determined (recall the effort difference of 1 and the reward/effort ratio of 5). Instead of the $\binom{100}{4} \times \binom{100}{20}$ entries formally required in the $Q = 20\%$, $P = 20\%$ quadrant of the payoff matrix, a more convenient notation is possible: only one entry is necessary to calculate the average payoffs (and similarly for the other three quadrants). This entry contains the payoff values for PASS and the payoff values for FAIL (in the given order), separated by a slash. Using this representation the payoff matrix is:

	P=20%	P=50%
Q=20%	5,0/0,5	5,−1/0,4
Q=50%	4,0/−1,5	4,−1/−1,4

The same representation can be used for the weighting matrix containing the probabilities of PASS and FAIL. The weighting matrix is:

	P=20%	P=50%
Q=20%	0.41/0.59	0.0625/0.9375
Q=50%	0.80/0.20	0.50/0.50

The payoff matrix and the weighting matrix can be multiplied in an element-wise fashion to get:

	P=20%	P=50%
Q=20%	2.05,0/0,2.95	0.3125,−0.0625/0,3.75
Q=50%	3.2,0/−0.2,1	2,−0.5/−0.5,2

And by pair-wise adding the payoffs for PASS and FAIL the following payoff matrix is obtained (after rounding off 0.3125 to 0.31 and 3.6875 to 3.69). Call this abstract 2×2 game TCTG:

	P=20%	P=50%
Q=20%	2.05,2.95	0.31,3.69
Q=50%	3,1	1.5,1.5

Please observe that this TCTG is essentially a Prisoner's Dilemma (as characterized by the fact that it has one N.E. which is not optimal for both players). It is not as nice and symmetric as the ITG; the differences come mostly from the fact that P(FAIL) takes irregular values, not precisely 0%, 50% or 100%.

References

[1] W.W. Gibbs. Software's chronic crisis. Scientific American, Sept. 1994, pp. 72–81 (1994).

[2] W. Poundstone. Prisoner's dilemma, Doubleday ISBN 0385-41567-2 (1992).

[3] J. Von Neumann. Zur Theorie der Gesellschaftsspiele, Mathematische Annalen, 100, pp. 295–320 (1928).

[4] J.F. Nash. Equilibrium points in N-person games, Proceedings of NAS (1950).

[5] M.J. Osborne, A. Rubinstein. A course in game theory, MIT Press (1994).

[6] E. Freeman. Building Gargantuan Software, Scientific American Presents, Vol. 10, N. 4, Special issue on extreme engineering, pp. 28–31 (1999).

[7] OSI. Conformance testing methodology and framework, Part 3: The Tree and Tabular Combined Notation (TTCN), ISO/IEC DIS 9646-3 (1990).

[8] S. Vuong, W. Chan, M. Ito. The OIUv method for protocol test sequence generation, In: Second International Workshop on Protocol Test Systems, Berlin, Oct. (1989).

[9] J.R. Moonen, J.M.T. Romijn, O. Sies, J.G. Springintveld, L.M.G. Feijs, R.L.C. Koymans. A two-level approach to automated conformance testing of VHDL systems, In: M. Kim, S. Kang, K. Hong (Eds.), IFIP TC6 International Workshop on Testing of Communicating Systems, Chapman & Hall, pp. 432–447 (1997).

[10] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence, Software – Concepts and Tools, 117:103–120 (1996).

[11] M.A. Nowak, R.M. May, K. Sigmund. The arithmetics of mutual help, Scientific American, June 1995, pp. 50–53 (1995).

[12] M.A. Nowak, K. Sigmund. Tit for tat in heterogeneous populations, Nature, Vol. 355, pp. 250–253 (1992).

[13] M.A. Nowak, K. Sigmund. A strategy of win-stay lose-shift that outperforms tit-for-tat in the Prisoner's Dilemma game. Nature, Vol. 364, pp. 56–58 (1993).

[14] M.A. Nowak, R.M. May. Evolutionary games and spatial chaos. Nature, Vol. 359, pp. 826–829 (1993).

[15] T. Gilb. Distinct software: a redundancy technology for reliable software, in: Infotech State of the Art Report on Software Reliability, Pergamon Infotech, Maidenhead, UK. (1977).

ENHANCED SYSTEM VERIFICATION AND VALIDATION, PERFORMANCE THROUGH METHOD INTEGRATION

J. van der Wateren[(1)], J.J. van den Berg[(1)], A.M. Bos[(2)], M.H.G. Verhoef[(2)], J. Kratz[(2)], E.J. Haverkamp[(2)]

[(1)]Chess Embedded Technology B.V., P.O. Box 5021, 2000 CA, Haarlem, The Netherlands, Email: Jeroen.van.der.Wateren@chess.nl, Jeroen.van.den.Berg@chess.nl

[(2)]Chess Information Technology B.V., P.O. Box 5021, 2000 CA, Haarlem, The Netherlands, Email: Bert.Bos@chess.nl, Marcel.Verhoef@chess.nl, Jeroen.Kratz@chess.nl, Erik.Haverkamp@chess.nl

ABSTRACT

In all phases of the lifecycle of the satellite, hardware as well as software development relies heavily on modeling and simulation. During the study phase, models of the major components are developed for proof of concept validation. During the design phase, simulation models are used to test and verify parts of the design. Those models can replace the environment or parts of the satellite hardware and software, which are momentarily not available. During integration, validation and verification models are used to check parts or sub-assemblies of the flight hardware and software. Those models are either environmental models or models resembling not yet available parts of the flight hardware and software. During the operational phase, simulation models are used in simulators for training the ground personnel and for testing new versions of on-board software before they are uploaded to the satellite.

Re-use of simulation models is limited. During the satellite lifecycle different design teams, who all have their own design methodology, concurrently develop the hardware and software sub-systems of the satellite. In time, a sub-system (hardware or software) also passes several design teams that vary over the development phases. And again, these design teams will often not re-use models developed by other teams. Although there is a great dependency during the whole lifecycle of a satellite on simulation models, these models are often created more than once for every sub-system and for every phase of a sub-system. The effectiveness and efficiency of today's verification and validation solutions can be greatly enhanced if this issue is properly addressed. We feel that the way forward is to better integrate currently available methods and techniques for verification and validation.

During the last year Chess worked intensively on a Guidebook that proposes the combination of three items: methods, techniques and SV&V facilities integrated into one single development strategy. This paper presents the results that the usage of this Guidebook has for the development of software in space projects.

1. DEVELOPING ON-BOARD SOFTWARE FOR SATELLITES

To obtain a development strategy in which methods, techniques, standardization, validation and verification are tightly coupled with the process of developing On Board Software, it is not necessary to invent new software development methods nor techniques used in the development process nor validation and verification facilities. The strength of our proposal consists of the combination of these three items: methods, techniques and SV&V facilities integrated into one development strategy. For the method part of the development strategy the ESA/ESTEC ECSS-E-40 software engineering standard is used, because this standard is already widely used in the development of On Board Software. Secondly a graphical language (UML), a formal language (VDM++) and their appropriate usage in the different software engineering phases of the ESA/ESTEC ECSS-E-40 standard are used. UML provides a way of easily communicating models to software engineers with different levels of knowledge. VDM++ provides a rigorous way of describing systems and offer validation and verification possibilities early in the development phase. To complete our approach with validation and verification the position of the Simulation HAndling Module (SHAM) and EuroSim models in the different development phases are defined.

1.1 VDM

VDM is a model-oriented specification language. This means that a specification in VDM consists of a mathematical model built from simple data types like sets, lists and mappings, along with operators which change the state of the model. VDM comes in two flavors: VDM-SL and VDM++. In VDM-SL the operations are specified as functions with expressions as their result and VDM++ is the object-oriented version of VDM in which the functional paradigm as well as the imperative (applicative) paradigm can be used. The authors chose VDM, because of their experience with this formal language and the possibility to use roundtrip engineering when combined with UML models.

1.2 UML

The Unified Modeling Language™ (UML) is an industry-standard language for specifying, visualizing, constructing, and documenting the components of software systems. Using UML, programmers and application architects can make a blueprint of a project, which, in turn, makes the actual software development process easier. Models are simplifications of reality and can therefore help people to better understand the system they are developing. The possibility to simplify the system is one of the powerful features of UML. UML can be used in phases A,B,C and D of the ESA software development process.

1.3 Simulation HAndling Module

During the past five years, SHAM has been used in many missions for Software Verification and Validation (SV&V). Examples of projects where the SHAM has successfully been used [1] are: ISO, SOHO, Cluster, XMM, Integral and Huygens. Recently SHAM has also been used in the Rosetta project, where it was the intention to even use it in the design phase to facilitate the software design engineer with a very powerful design and debugging tool.

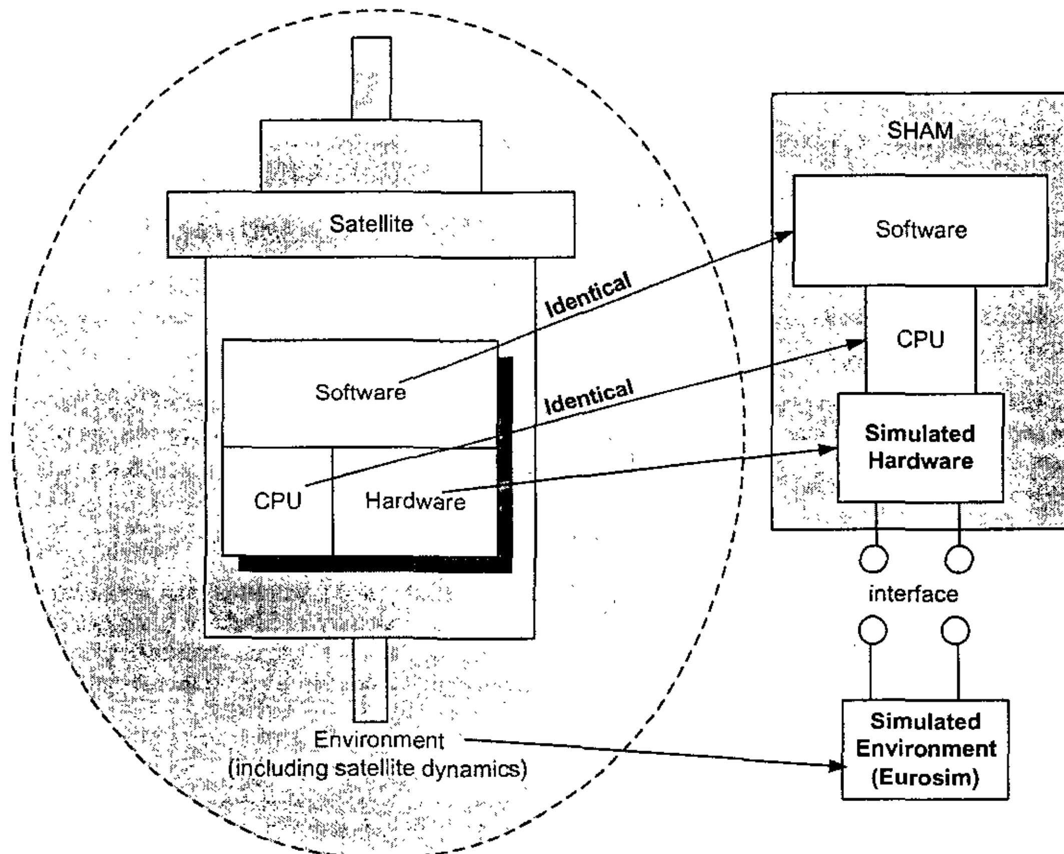


Fig. 1. SHAM, all white squares are original parts of the target system. This means the SHAM contains the real CPU and runs the real software (OBS). Simulated items are presented with dotted squares like the hardware and environment.

The SHAM is a hardware emulator that is able to control the execution of the developed On Board Software (OBS) on the *identical* CPU that will be used in the satellite. The SHAM controls this CPU and simulates all the hardware around the CPU, as can be seen in Fig. 1. By not simulating the CPU but using the real CPU, the validation results of the OBS-tests are very reliable. There is only one single version of the OBS that is used operationally and tested to exclude differences in software behavior between debug (or test) and operational releases. The OBS can be tested non-intrusively which means that no debug statements have to be compiled into the binary. The developer and the user are certain that the OBS that is tested on the SHAM is exactly the same as the OBS running on the real satellite in space. Another advantage of the SHAM is the fact that the CPU does not have to be emulated in software. Every year CPU's become faster and more complex which makes the software emulation of these CPU's in real time more difficult.

1.4 EuroSim

Along with the SHAM, different kind of simulators, like EuroSim, Simula and MatLab are used to run simulation models needed during the development of the OBS for satellites. These simulators are able to simulate the satellite and its environment in space. Chess recently developed a general SHAM Simulator Interface (SSI) for the Eurosim platform to make it possible to run the real OBS on the real CPU in a simulated space environment. The SSI will in the future also be available for other simulation platforms.

2. THE INTEGRATED VERIFICATION AND VALIDATION APPROACH

The software development process can be decomposed into phases. Every phase results in one or more products that is an input to a following phase. This process is described by the well-known V-model (Fig. 2).

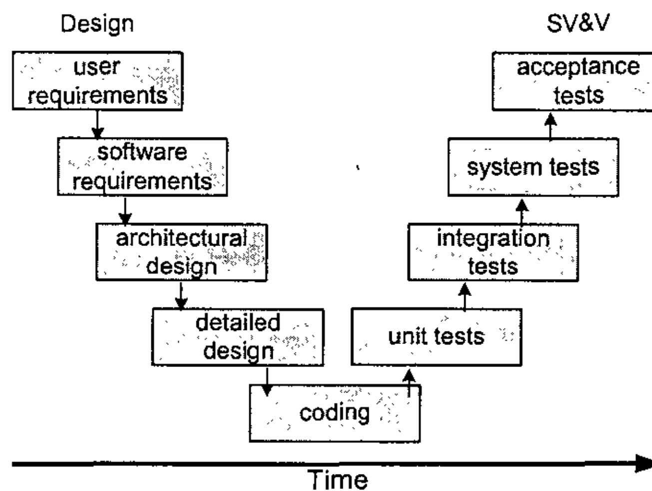


Fig. 2: V-model, relation between design- and SV&V phases as a function of time.

The authors extend on this V-model by defining the mirrored Y-model as is shown in Fig. 3. This model presents the proposed development strategy , combining the strength of:

- Development techniques (VDM-SL, VDM++, UML)
- SV&V facilities (EuroSim, SHAM)
- Model evolution (OBS, Environment)

In the V-model the time line is from left to right, in the mirrored Y-model the time flows from top to bottom and presents the different ESA development phases (A-F). The V-model represents only the development phases, while blocks in the mirrored Y-model denote the documents belonging to the phases. Note also the model development on the right side of the model, there is a continuous enhancement/reuse of the models.

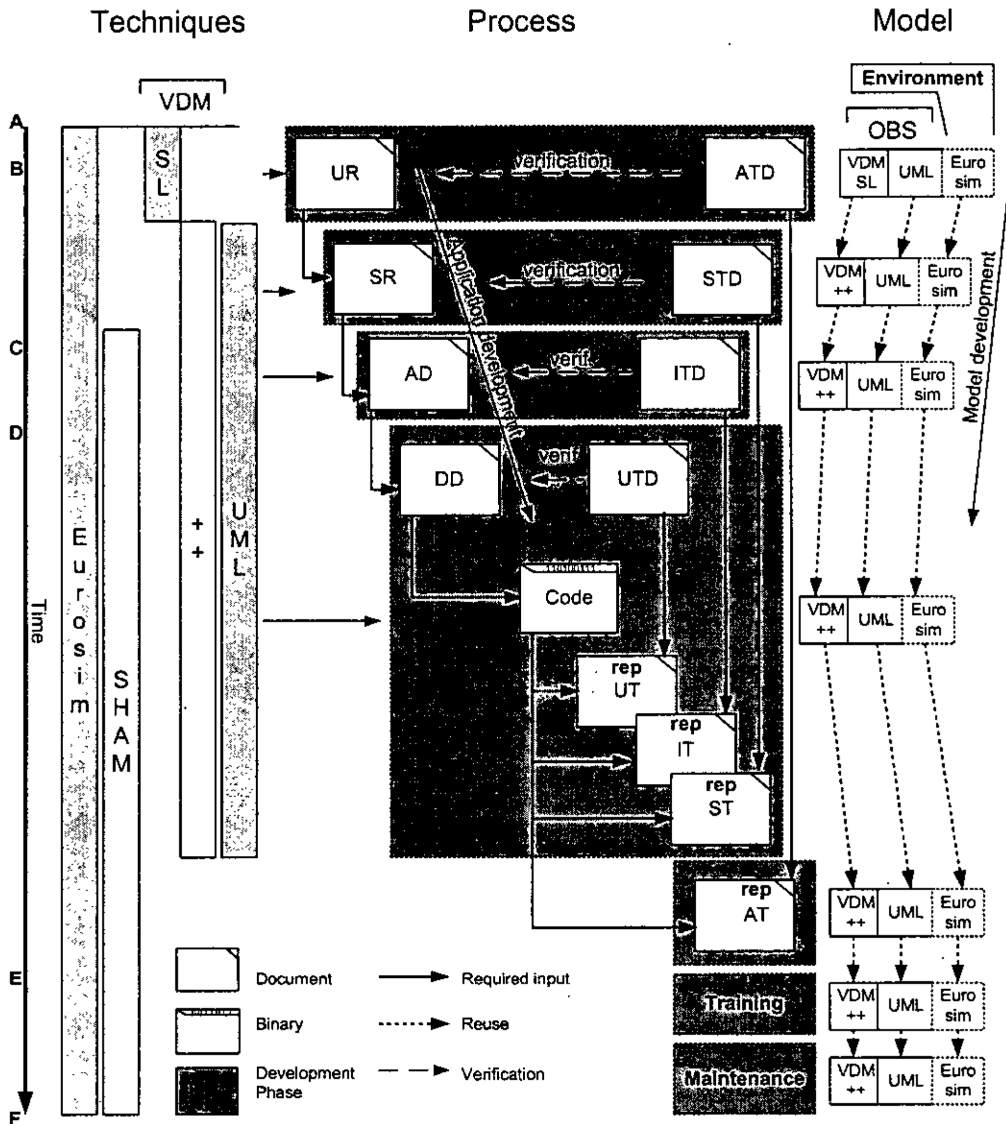


Fig. 3: The mirrored Y-model; the time flows from top to bottom and presents the different ESA development phases (A-F), the blocks represent the documents belonging to the development phases.

The mirrored Y-model emerged during the writing of our Guidebook. This Guidebook describes the integrated software development process as presented in the mirrored Y-model. It proposes a unified (space onboard) software development approach in which errors are detected as early as possible in the software development and by which errors will not even reach the review meeting. It suggests a combination of existing methods in such a way to improve reuse from phase to phase and to improve the integration of validation & verification in the development process. The model shows vertically on the left-hand side the tools and techniques that are used together with the ESA software development phases (A-F). On the right-hand side the evolution and reuse of the different models is presented. For the OBS this is done by showing the UML and VDM models and for the environment the reuse of the EuroSim simulation models is described. In the middle of the mirrored Y-model the software development process, as known from the V-model, divided into the different phases is shown. Horizontally the Validation and Verification part of the process is presented.

3. PMC-CASE: OBS DEVELOPMENT CASE-STUDY

To obtain some practical experience with the proposed development strategy, the authors performed a simplified test-case. The main goal of this case was the development of the On Board Software (OBS) for an ideal satellite. This satellite was defined as a Point Mass of 100 kg, moving in a 2-dimensional world. Since we were developing the OBS to control this point-mass we named the satellite: Point-Mass Controller (PMC). The PMC was equipped with two thrusters, both having their resulting forces in the opposite direction. The thruster for the positive direction could apply a force of 200 Newton and the other thruster 50 Newton, the only sensor available to the PMC was an acceleration sensor. The OBS should be able to move the PMC over a distance of 100 meter by controlling the thrusters and reading the acceleration sensor. The remainder of this paper the PMC-case is used to explain the ideas in the different phases of the mirrored Y-model as described in the Guidebook.

[Phase A - B]
During the concept phase the requirements of the OBS are to be defined. At first glance it seems a simple case, therefore it shouldn't be that difficult to find the user requirements. The authors used Matlab to model the physics of the PMC in order to get a feeling about the system and to make it possible to try some different scenarios. The first requirements were defined, for example:

- The two thrusters are not allowed to be switched on at the same time
- The PMC is not allowed to pass the final position of 100m, it should move smoothly to the final position without any overshoot.

After the first requirements were defined, the system was described in the formal specification language VDM-SL. This formal language makes it, among other powerful features, possible to add pre- and post-conditions to the system model. Definition of pre- and post-conditions is the part of VDM that is very powerful and will most likely give important discussions among the designers and the customer. In order to determine these conditions, the designer has to have a good understanding of what the system will do. The designer will come up with questions that he otherwise would not have come up with[1]. In VDM-SL the designer can execute his model to test it in several ways. Since we are now in the requirements definition phase we should already define the acceptance tests that should be performed in one of the final phases to test if all the requirements are satisfied. The authors defined the acceptance tests in VDM-SL and could execute them to test the requirements! By now no system is build and no code has been written, but the designers are already able to test the modeled system they want to build.

The Object Oriented (OO) approach was chosen for the continuation of the PMC-case, therefore the VDM-SL specification is converted to VDM++. One of the strengths of VDM++ is the ability to perform round-trip engineering with UML. The VDM++ toolkit has been used to automatically generate a UML-class diagram of the system. This class-diagram provides the designer/developer with an easier to understand description of the system. The VDM++ specification and UML-class diagrams were now changed into a more detailed and complete specification that was used as an input for the architectural design phase [Phase C]. In order to verify the specifications, more detailed test-cases were developed in VDM++. For some parts of the software it was necessary to create prototypes, this was done by executing this code in the environment model. By this time the designers had to make the architectural design and this was done by creating sequence diagrams and state diagrams that are also part of UML. Interfaces were described and the architectural design was completed.

[Phase D]
The developers were given the architectural design of the OBS in order to start the code-design. For the implementation of the software the developers used the SHAM/EuroSim combination. EuroSim used the environment model from Matlab that the designers developed in one of the early phases of the PMC-case. After the code was written the test-cases as described in VDM-SL and VDM++ were used to test the OBS. Chess has coupled the SHAM to EuroSim by designing and implementing a SHAM Simulator Interface (SSI). This interface allows the user to connect the SHAM and the Simulator (EuroSim) forming one powerful system for simulating and testing (onboard) software in an early stage of a project. The PMC-case was used to initially test the SSI and check its abilities to intertwine the two systems. This action proved the powerful abilities the SSI has to offer.

[1] The authors experienced this themselves during the PMC-case.

4. CONCLUSIONS AND FUTURE WORK

This paper presented the status of our ideas about an integrated OBS development process. We found that considerable improvements can be obtained by mutually tuning existing methods and techniques.

- The software development methodology and graphical language UML have been coupled [6] to ECSS-E-40.
- A formal specification language was added to UML, such that the specifications, design and software production could be formalised. The combination of UML and VDM enable a smooth transition from requirements, design to code generation.
- The SHAM was already in use for OBS development. In this project we positioned the environment model from the SHAM environment to EuroSim, by which reuse of environment models could be obtained from phase to phase. Integration with operation simulators is now also feasible, as is shown by [5]

This combination learned us that validation and verification became effective over all development phases: requirements could be formally verified, the design could be verified with respect to the requirements and the software with respect to the design and the requirements.

The development tools and techniques (VDM-SL, VDM++, UML, EuroSim) were chosen by the authors, because of their experience and knowledge of these particular tools. This does not imply that other similar tools can not be used. The Guidebook proposes a software development process and does this in such a way that any suitable tool for a particular problem can be used.

The project proceeds in 2001 with a more solid example and the implementation of a coupling between the SHAM and EuroSim.

5. REFERENCES

1. A.M. Bos, J. van der Wateren, *"Real-Time Software Testing throughout a Project Life-cycle using Simulated Hardware"*, Simulators for European Space Programs, ESTEC, Noordwijk, 1998.

2. MS. Mejnertsen, K. Hjortnaes, S. Ekholm et al, *"Software Validation Facility for the SPARC Micro-processor"*, Data Systems in Aerospace, ESA SP 447, Lisbon, 1999

3. A.M. Bos, M. Geerling, M.H.G. Verhoef, J. van der Wateren, *"Development of Safety-critical Software using Formal Methods in IDE"*, International Aeronautic Federation, Amsterdam, 1999

4. M. Brouwer, A.A. Casteleijn, H.A. van Ingen Schenau et al, *"Developments in Test and Verification Equipment for Spacecraft"*, Simulators for European Space Programs, ESTEC, Noordwijk, 2000

5. L.J. Timmermans, T. Zwartbol, B.A. Oving, A.A. Casteleijn, M.P.A.M. Brouwer, *"From Simulations to Operations: Developments in Test and Verification Equipment for Spacecraft"*, DASIA, Nice, 2001

6. B.A. Oving, L.J. Timmermans, A.A. Casteleijn, T. Zwartbol, A.M. Bos, M.H.G. Verhoef, J. van der Wateren, *"Efficient Development and Validation of Spacecraft Avionics Through Improved Tool Integration"*, Toulouse, IAF 2001, to be published.

FROM SIMULATIONS TO OPERATIONS: DEVELOPMENTS IN TEST AND VERIFICATION EQUIPMENT FOR SPACECRAFT

L.J. Timmermans, T. Zwartbol, B.A. Oving, A.A. Casteleijn, M.P.A.M. Brouwer

National Aerospace Laboratory NLR, Space Division
P.O. Box 153, 8300 AD, Emmeloord, The Netherlands
Phone: (+31) 527 24 8444 Fax: (+31) 527 24 8210
E-mail: {timmerlj, zwartbol, oving, castelyn, mbrouwer}@nlr.nl

ABSTRACT/RESUME

Supported by Research and Development programs of the European Space Agency (ESA), simulation and test tools are being developed to improve the life cycle cost efficiency of spacecraft development. Standardisation and rationalisation, use of (C)OTS products, reuse of both software and hardware are some of the lines along which schedule optimisation and cost reduction is pursued.

Based on experience with the development, production and use of test equipment for scientific satellites such as XMM-Newton and INTEGRAL, the National Aerospace Laboratory NLR is developing a next generation of Test and Verification Equipment (TVE) for spacecraft avionics systems, such as Attitude and Orbit Control Subsystems (AOCS). Starting points for these developments are the application of existing relevant technologies, modularity, scalability, commonality and reuse of tools, equipment and results during the various phases of the spacecraft life cycle.

This paper will focus on the interface between TVE and ESA's latest generation Spacecraft Control & Operation System, SCOS-2000, as important element of the next generation TVE to enable reuse of technologies from simulations to flight operations.

1. INTRODUCTION

During spacecraft development, a number of simulation and test facilities are used to support different activities like mission analysis, design and development, Assembly, Integration and Verification (AIV) and flight-operations preparation and training. Simulation is used extensively in design verification and operations preparation, but simulation is also required for test benches involving real flight hardware. Electrical Ground Support Equipment (EGSE) is used to verify the spacecraft's functionality on ground, by stimulating it with test signals and telecommands (TC), and analysing its responses via monitoring interfaces and telemetry (TM).

From past spacecraft programs, a number of important issues for the optimisation of the spacecraft life cycle have emerged, e.g.:

- Need to reuse simulation environments and simulation model software, not only during the development and verification phases, but also during commissioning and in-flight operations.
- Need for early on-board software prototyping and validation. Historically much simulation effort was spent on the verification of control algorithms functionality and performance, before implementation in the On-Board Computer (OBC). Experience has shown that it is equally important to exercise the (often very complicated) Failure Detection, Isolation and Recovery (FDIR) functions implemented in on-board software of autonomous spacecraft, and possibly associated operational control procedures in an early stage of the development. The use of a simulation facility, possibly coupled to an OBC emulator, will enable early prototyping and validation of control algorithms and autonomy functions.
- Need for the development of operational flight control procedures as early as possible, such that the on-board software and operational procedures are exercised to the greatest possible extend on the ground. The developed operational procedures shall be usable during system level integration and test, commissioning and operations.
- Need to use a (central) spacecraft database (SDB) throughout the life cycle. As the life cycle consists of several phases with activities taking place at different locations, it shall be possible to interface to, build up and use the SDB in the different phases and at different places. This requires compatibility and import/export capabilities of database tools used.

Based on experience with production and use of test systems for satellites such as XMM-Newton and INTEGRAL, NLR is developing a new generation of Test and Verification Equipment (TVE) for spacecraft avionics systems. Starting points for the developments are: the existing TVE technology, lessons learned from XMM-Newton and INTEGRAL, application of relevant technologies developed in ESA -R&D- programs, commonality, modularity and scalability, reuse during the various phases of the spacecraft life cycle, use of COTS products.

2. INTEGRATION OF RELATED ESA TECHNOLOGIES

Among the related technologies being developed under ESA programs, are the Project Test Bed (PTB), the Spacecraft Control & Operation System (SCOS), the Software Validation Facility (SVF), and the TVE itself. These technologies are independently used in different phases of the spacecraft life cycle. By reuse of these technologies, the next generation TVE bridges the spacecraft life cycle [2].

The Project Test Bed is intended to be used as a single test platform, based on real-time simulation, that is able to provide support to the different phases of a project, and be reused across projects [3]. The core of the test bed is the real-time simulator that includes libraries of both spacecraft subsystem models and environment models. The simulator is based on the real-time simulation environment EuroSim [4], which provides support for model development and integration, simulation execution and analysis of results.

SCOS-2000 is the latest generation Spacecraft Control & Operation System [8], based on long-time experience, developed and used at the European Space Operations Centre (ESOC). The system is scalable in order to suit different mission requirements, budgets and/or mission phases and provides the essential functions to monitor and control a satellite both in orbit and during testing.

The Software Verification Facility is an onboard software test environment, which provides representative behaviour for a target system, based on the Simulation HAndling Module (SHAM), an onboard computer (OBC) emulator, developed by Chess Engineering B.V. [5]. SVF also comprises a software environment for a/o TM/TC handling and simulation. The SVF/SHAM concept has widely been used the last few years for several ESA scientific satellites.

The Test and Verification Equipment has been developed by NLR for integration and testing of spacecraft avionics systems. Next to a real-time simulator, TVE features a generic front-end with a modular, VME based, architecture, containing hardware interfaces to the spacecraft data buses and to the avionics units for stimulation and monitoring.

While PTB is currently mainly used in the first phases of the spacecraft life cycle, SCOS is used in the final phase. TVE is typically used in phase C/D, see Fig. 1. The next generation TVE bridges the life cycle by integrating a/o the EuroSim and SCOS-2000 technologies. Integration of SHAM, used for both software development and software maintenance, is taking place. The integration of these technologies will stimulate the reuse of simulation and test tools and project results from the early simulations phases (A/B) to the final operations phase (E).

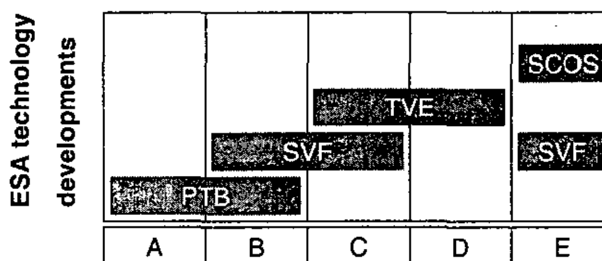


Fig. 1. Overview of TVE related ESA technologies in the spacecraft life cycle

3. THE TVE CONCEPT

TVE was originally developed for ESA, and is (being) used for testing of the AOCS of the XMM-Newton and INTEGRAL scientific satellites, at AOCS subsystem and spacecraft system level [1].

TVE is a closed loop test bench facility (no real motion) for avionics systems. A complete avionics system, together with dynamics and environment can be considered as a loop, which is actively closed by the Attitude Control Computer (ACC) or, more general, an On-board Computer (OBC). The OBC cyclically reads out the sensors, performs the control and other tasks (e.g. FDIR, TM/TC) and issues the resulting commands. The spacecraft dynamics and environment are simulated. The sensor electronics are stimulated by the simulation such that they produce the measurements for the OBC. The OBC wil command the actuators to control the spacecraft dynamics. The relevant signals from the actuator

units are acquired by monitoring interfaces and routed back into the simulation. The dynamics simulation runs at a fixed simulation cycle rate, which generally is a multiple of the OBC sample frequency.

During the Assembly, Integration and Verification activities in phase C/D, the avionics subsystem is gradually built up, depending on the schedule of incoming units. Hardware units not yet present are to be simulated. The functional behaviour of these units is simulated in software, while the units physical interfaces are simulated by a hardware unit simulation interface. In this way verification can be performed with any combination of real and simulated units; starting from pure software simulations, via integration of a single OBC, gradual replacement of software models by hardware units, up to a fully integrated subsystem.

Four main elements can be identified in test and verification of spacecraft, see Fig. 2:

1. The System Under Test (SUT), which is (part of) the spacecraft avionics systems that needs to be tested as if it is in its operational environment; an important unit is the OBC that controls the spacecraft
2. Front-end Environment (FE). The Front-end electronics consist of two parts:
   - stimulation and monitoring equipment, which electrically or physically stimulates sensors and electrically monitors actuator units
   - data bus interfaces, which are able to:
     - simulate missing units (address, data interface)
     - monitor all traffic (data, instructions) on the bus
     - simulate the bus controller
     - perform fault injection
3. Simulation Environment (SE), which hosts the Simulation Model Software (SMS) to simulate the spacecraft dynamics and space environment, to calculate stimuli values, and to simulate avionics units
4. Checkout Environment (CE), which contains the knowledge about the SUT (e.g. procedures and TM/TC database) and from which automatic test procedures are executed and AIV activities are controlled

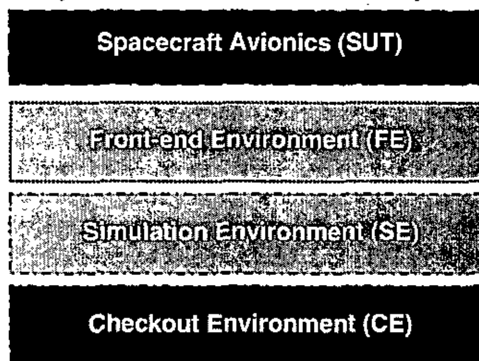


Fig. 2. Main elements in test and verification of spacecraft avionics

The term environment is used as a combination of software and hardware.

Front-end Environment (FE)

For the current TVE FE, data bus interfaces are available for Modular Attitude Control Systems (MACS) and On Board Data Handling (OBDH). For the next generation TVE, new developments are test interfaces for the MIL-1553 bus, the new PSS-04-255 standard OBDH bus, and the SpaceWire (upgraded IEEE-1355) data link. TVE's electrical stimulation and monitoring equipment interface comprises a set of low level boards. For stimulation, NLR has developed Analog Stimuli Interface (ASI) and Bi-level Stimuli Interface (BSI) boards which provide configurable, high-resolution current/voltage stimuli channels. These channels have fully isolated test interfaces with overvoltage and overcurrent protection. For acquisition of data to be monitored, Monitoring Interface (MOI) and General Timing Monitor (GTM) boards have been developed. With the MOI board analogue and bi-level values can be acquired. The GTM board provides timers for the measurement of time duration with 1 ms resolution. For the next generation TVE also features an IRIG-B timer board. Furthermore, dedicated, manufacturer supplied, Units Checkout Equipment (UCE) can be integrated in the FE, e.g. for operation, stimulation and monitoring of star tracker or sun acquisition sensor units.

Simulation Environment (SE)

The current TVE is based on ProSim, a general purpose simulation tool and ancestor of EuroSim. For the next generation TVE, EuroSim will be used, which is a configurable simulator tool that is able to support all phases of space and non-space programmes through real-time simulations with a person and/or hardware-in-the-loop. For PTB, the simulator is also based on the EuroSim real-time simulation environment. TVE-specific tools and interfaces will be developed as extensions to EuroSim. has the possibility to interface with other processes via the so-called external simulator access.

Checkout Environment (CE)

Within the current TVE, the checkout environment to control the tests is strongly coupled with the ProSim. For system level testing, there are two ways to develop and operate the avionics subsystem related tests:

- use TVE, and request a Central Checkout System (CCS), also referred to as core EGSE, to send telecommands to the spacecraft. In this case, both by the CCS and TVE must process telemetry.
- use the CCS, and remotely control the TVE simulation environment and test interfaces.

For XMM-Newton, TVE was used as Checkout Environment. All tests have been developed in the ProSim/EuroSim Mission Definition Language (MDL), a C-like language with access to (part of) the simulator. Because MDL is an interpreter language, test scripts could also be changed during a test. Also flight procedures have been implemented in MDL, for verification.
For INTEGRAL the CCS is used to perform all system level tests, which has the advantage that telemetry only needs to be processed once, and all AIV engineers use the same checkout environment. On the other hand, tests developed at subsystem level with the TVE (on also the system level tests from XMM) can not fully be reused.
Since SCOS-2000, with its EGSE capabilities, will be used as Checkout Environment for the next generation TVE, test procedures can be reused at avionics subsystem, spacecraft system, and spacecraft operations level.

The next generation Test and Verification Equipment will improve life cycle cost efficiency of spacecraft development, especially of ESA scientific spacecraft. Reuse of existing technologies and standardisation will bridge the spacecraft life cycle from the early simulation phases (A/B) to the final operations phase (E). These technologies include SCOS-2000 for checkout and operations, EuroSim for simulation such that models developed with PTB can be reused, and the TVE front-end to interface with the flight hardware. Furthermore Chess BV (NL), supported by NLR, is developing and interface between the SHAM and EuroSim [12], which will facilitate the reuse of the SVF/SHAM technology. The reuse of these technologies is shown in Fig. 3.

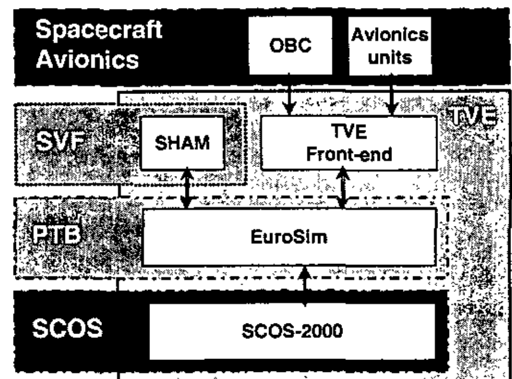


Fig. 3. Reuse of existing ESA technologies for main elements of the next generation TVE

4. COMMONALITIES BETWEEN SPACECRAFT CHECKOUT AND OPERATIONS

Systems for spacecraft checkout and spacecraft operations have a large degree of commonality. The system used to perform spacecraft checkout is traditionally called Electrical Ground Support Equipment (EGSE) and is used for Assembly, Integration and Verification (AIV) at both spacecraft system level and spacecraft subsystem level (like AOCS). During system level tests the EGSE controls a number of Special Checkout Equipment (SCOE), which are similar to the subsystem EGSE but with different functionality. The EGSE enables to control behaviour of spacecraft equipment under test by sending (tele)commands and acquiring signals and telemetry through specific interfaces. The system used for mission preparation and spacecraft operations is called Mission Control System (MCS). As for the EGSE, the MCS has to deal with spacecraft monitoring and commanding.

ESA has a long-standing objective of having a common system to be used as an EGSE and later on as an MCS [6, 7]. From a technical viewpoint it is obvious that for those systems many functions are similar, if not the same. However the necessary harmonisation had not in practice been possible, since these two systems are used in different phases of the mission and, moreover, under different responsibilities; the spacecraft prime contractor for the EGSE and ESA/ESOC for the MCS. This problem is essentially of a managerial and a contractual nature. By merging the Technical Directorate of ESTEC with the Directorate of Operations at ESOC into a single Directorate Technical and Operation Support (TOS) the EGSE support and operation support (including MCS) are now under the same technical responsibility. The combined system is called generically "EGSE and Mission Control System" (EMCS). The ESA Scientific Directorate is strongly pushing the spacecraft primes to propose systems that are common to EGSE and MCS.

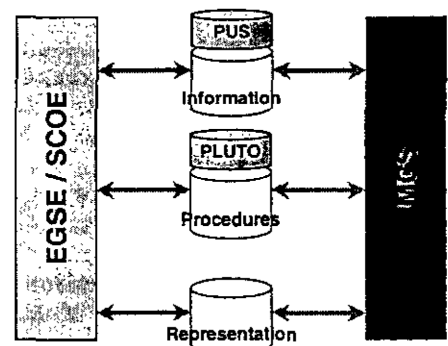


Fig. 4. EGSE / SCOE and MCS systems use common data

The data used by an EGSE and a MCS are very similar, see Fig. 4. Both the EGSE and the MCS must have knowledge about the spacecraft and its behaviour. This is described in a database that is often called the Mission Information Base (MIB). The core information in the MIB is the Telemetry and Telecommand characteristics. By using the ESA Packet Utilisation Standard[1] (PUS), missions can easily reuse the same EGSE or MCS infrastructure.
Both the EGSE and the MCS are operated using procedures. During checkout, activities are described using a formal test language. During operation one has flight operation procedures which could also be defined using an operation language. By using a common language, like the Procedure Language for Users in Test and Operations (PLUTO) as defined in the ECSS-E-70-32 standard [10], test and operational procedures can easily be reused.
The last type of data needed by both an EGSE and a MCS are display data, especially TM/TC related mission representation, which can be alphanumeric displays, graphics or synoptic/mimic displays. By using the same representation, people involved in the AIV phase can easily provide support during the (early) operations, and vice versa, resulting in a better knowledge transfer and improved operations.

An important difference is that the EGSE needs to control Special Checkout Equipment (SCOE), which can in essence be treated like spacecraft TM/TC. Note that TVE can be used both as standalone EGSE during subsystem level tests, and as SCOE during system level tests. Another important difference is that the EGSE/SCOE tries to uncover weakness (e.g. through error injection and checks) of the spacecraft, whereas the MCS has to work around such faults in order to keep the spacecraft alive.

From a system level, however, the interface of the (core) EGSE with the several spacecraft SCOEs can be compared with the interface of the MCS with the several ground stations. In both cases an interface is necessary to exchange the data in a standard way. In case of SCOS as MCS, the Network Control and Telemetry Routing System (NCTRS) is used as interface, see Fig. 5.

Note that for EGSEs and MCSs, one can distinguish two types of commonality: the horizontal commonality where the same system is reused between different missions, and the vertical commonality where for a same mission the same system is reused between the AIV phase and the operations phase.

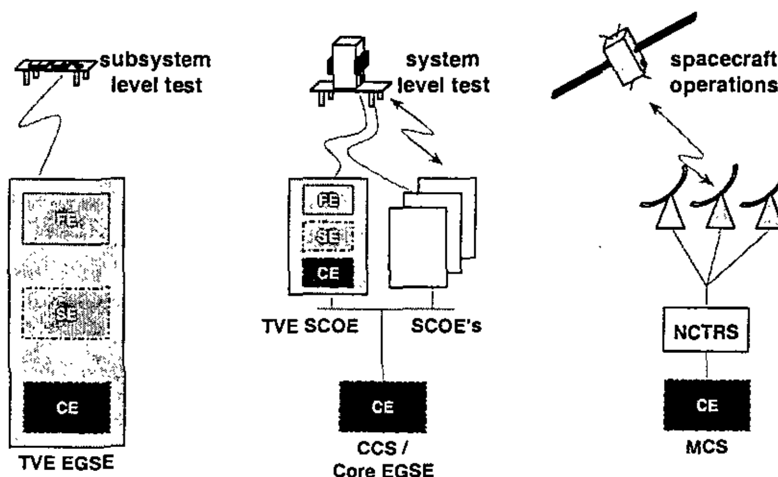


Fig. 5. System commonality for avionics subsystem level tests, avionics tests at system level and spacecraft operations

5. THE INTERFACE BETWEEN SCOS-2000 AND TVE

During operations, the SCOS-2000 has the role of MCS and as such controls the spacecraft via TM/TC. During checkout, SCOS-2000 has the role of CE, controlling both the SUT and the TVE (and other EGSE/SCOE equipment). The interface between SCOS-2000 and TVE will be TM/TC based, such that for SCOS-2000 a common interface is maintained.

Internally, TVE components communicate via the TVE Message Transfer Protocol (TMTP). The TMTP is mainly used for communication between CE/SE and FE. The TVE Protocol Messages (TPM) contain e.g. data for the electrical and data interfaces, status info and error/warnings.

The conversion between the TM/TC based (external) SCOS-2000 communication and the TMTP based (internal) TVE communication will be provided be a Routing Environment (RE). Telecommands from SCOS-2000 to TVE will have an embedded TPM, which is passed by the RE. The RE will put data from TVE in telemetry packets, to be read by SCOS-2000. All communication with the SUT will be passed through the RE in an appropriate format. Effectively, the

[1] The PSS-07-101 standard will be replaced by the ECSS-E-70-41 standard [11]

data stream coming from SCOS-2000 is split into a TPM stream for the TVE and a TM/TC stream for the SUT. The conceptual architecture is shown in Fig. 6.

Since SCOS-2000 in the role of MCS interfaces with the NCTRS, an RE has been developed that implements the NCTRS ICD [9] on CE side, thus acting as an NCTRS simulator during checkout. On SE side, the EuroSim External Simulator interface has been implemented. For further reuse the RE should be able to connect different types of CE and SE, other than SCOS-2000 and EuroSim.

The TVE related TM/TC includes:

- control and monitor of the simulator state
- control and monitor of the real-time simulator test environment (Mission Definition Language actions)
- control and monitor of the simulation model variables (data dictionary)
- any TPM to TVE internal processes, e.g. to perform error injection
- error and warning messages

The user is provided with dedicated displays to command and monitor the TVE from the SCOS-2000 checkout environment.

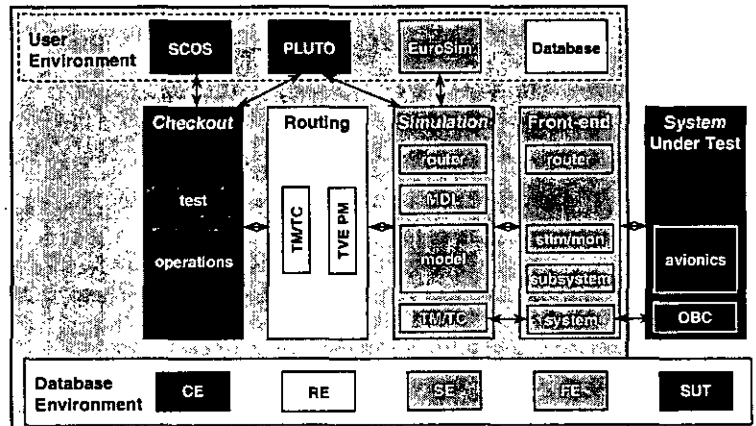


Fig. 6. Conceptual architecture of the next generation Test and Verification Equipment, showing the main elements and interfaces

6. CURRENT STATUS

Currently, the next generation Test and Verification Equipment at NLR operates in the following configuration:

- SCOS-2000, release 2.0, running on a Sun platform, as Checkout Environment
- EuroSim Mk2, running on a Linux PC, as Simulation Environment
- TM/TC based interface between SCOS-2000 and TVE, with a NCTRS simulator implementation as Routing Environment

For the target configuration TVE, a Linux version of SCOS-2000 will be used. The Linux version will become available in the next months.

Fig. 7 shows a screen snapshot that reflects the current development status, with part of the CE, RE and SE operational with the Point-Mass Controller (PMC) test case. The PMC-case is developed by Chess as case study for the development of On-Board Software [12]. A point-mass is commanded to a certain position with the OBS controlling two thrusters (in opposite directions and unequal forces). In the NLR configuration the OBS is running in the FE. Newton's law is simulated in EuroSim, which provides only the acceleration data as sensor for the PMC. The graph shows the position of the point-mass in time. The target position is set with a telecommand from the SCOS-2000 Manual Stack window, which shows two telecommands for the TVE system. The NCTRS simulator window shows the connections to SCOS-2000 and a hex dump of a raw telecommand.

7. CONCLUSIONS

NLR is actively contributing to the trend of maximising reuse of software and hardware throughout the spacecraft life cycle. It is developing modular and scalable Test and Verification Equipment, which integrates PTB/EuroSim, SVF/SHAM and SCOS-2000 technology, such that it can be (re)used for simulation and verification from the early simulations to final operations. Standardisation of tools is pursued.

8. ACKNOWLEDGEMENTS

NLR thankfully acknowledges the fruitful discussions with the ESA Technical Operations and Support Directorate, especially the Mission Control System Division, the Test and Operations section, and the Modelling and Simulation section. Also the useful feedback from the current TVE users - Astrium Friedrichshafen for XMM-Newton and Alenia Turin for INTEGRAL – for their suggestions in improving the Test and Verification Equipment is much appreciated.

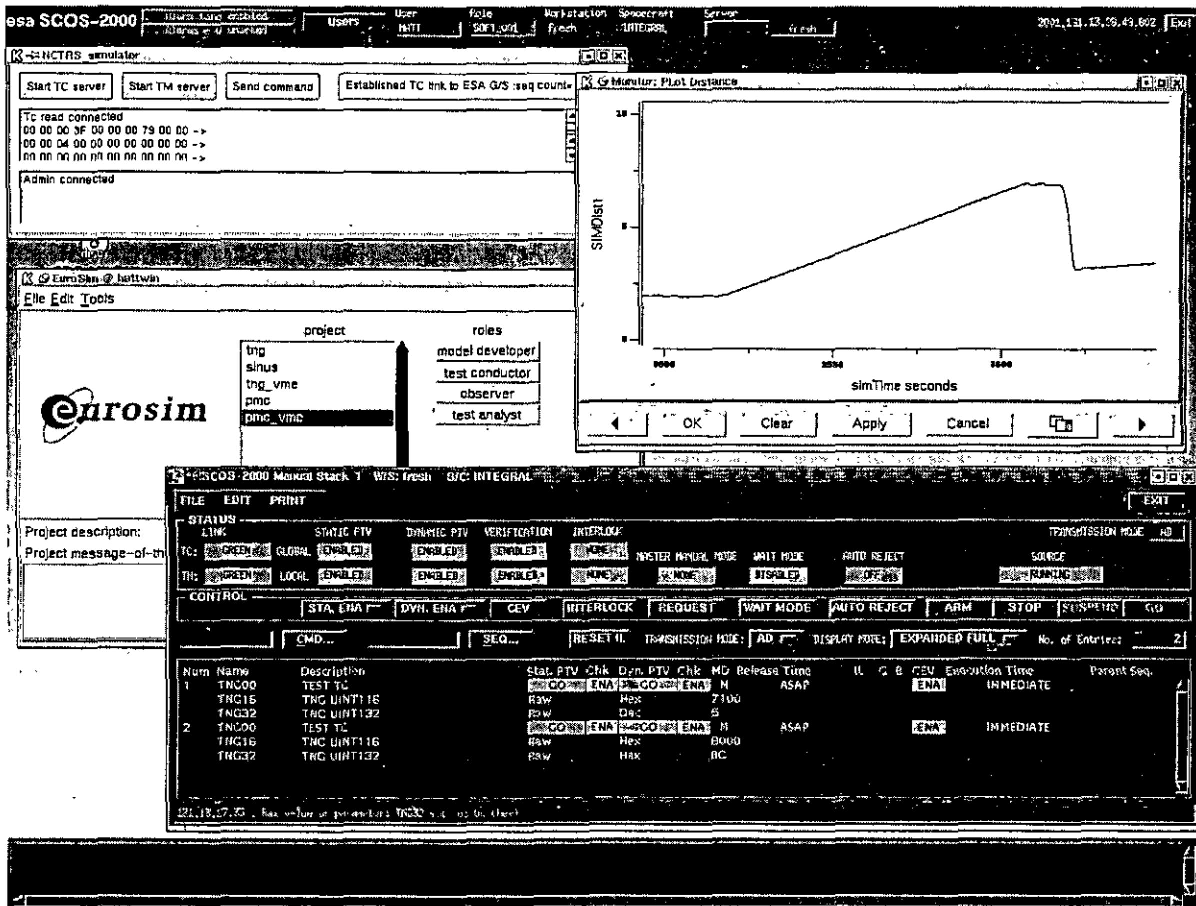


Fig. 7. Screen snapshot reflecting the development status for the next generation TVE, with user interfaces of SCOS-2000, NCTRS simulator and EuroSim operational in the Point-Mass Controller test case.

9. REFERENCES

1. H.A. van Ingen Schenau, L.C.J. van Rijn and J. Spaa, *Test and Verification Equipment for the Attitude & Orbit Control System of the XMM satellite*, NLR-TP-98236, Athens, DASIA 1998.
2. M.P.A.M. Brouwer, A.A. Casteleijn, H.A. van Ingen Schenau, B.A. Oving, L.J. Timmermans, T. Zwartbol, *Developments in Test and Verification Equipment for spacecraft*, Noordwijk, SESP 2000.
3. R.Franco & J. Miró, *The Project Test Bed and its application to future missions*, ESA Bulletin n.95, August 1998.
4. D.J. Schulten, U.G. Termote, M.J.H. Couwenberg, *EuroSim and its applications in the European Robotic Arm Programme*, Montreal, DASIA 2000.
5. J. van der Wateren, A.M. Bos, *Real-time software testing throughout a projects life cycle using simulated hardware*, Noordwijk, SESP 1998.
6. J-F. Kaufeler, *ESA management approach for a common EGSE and MCS system*, Toulouse, SpaceOps 2000.
7. J-F. Kaufeler, B. Melton, M. Jones, *Spacecraft check-out and flight control systems: compatibility or commonality*, Tokyo, SpaceOps 1998.
8. SCOS-2000 Team, Terma, *SCOS-2000 System Level Architectural Design Document*, S2K-MCS-ADD-0001-TOS-GCI, Issue 3.2, 22/10/2000.
9. C. Lannes, *Interface Control Document, NCTRS, Volume 2- Detailed interface definition: MCS*, N2K-MCS-ICD-0002-TOS-GCI, Issue 2.0, 15-Nov-1999.
10. ECSS-E-70-32 Ground systems and operations - Procedure definition language
11. ECSS-E-70-41 Ground systems and operations - Telemetry and telecommand packet utilization.
12. J. van der Wateren, J. van den Berg, A.M. Bos, M.H.G. Verhoef, J. Kratz, E. Haverkamp, *Enhanced System Verification and Validation Performance through Method Integration*, Nice, DASIA 2001, to be published.

invensys
Software Systems

BAAN

Service pack testing
in a commercial development environment

B.H.J. (Ben) van Buitenen
Manager Product Testing Group ERP
Baan Development

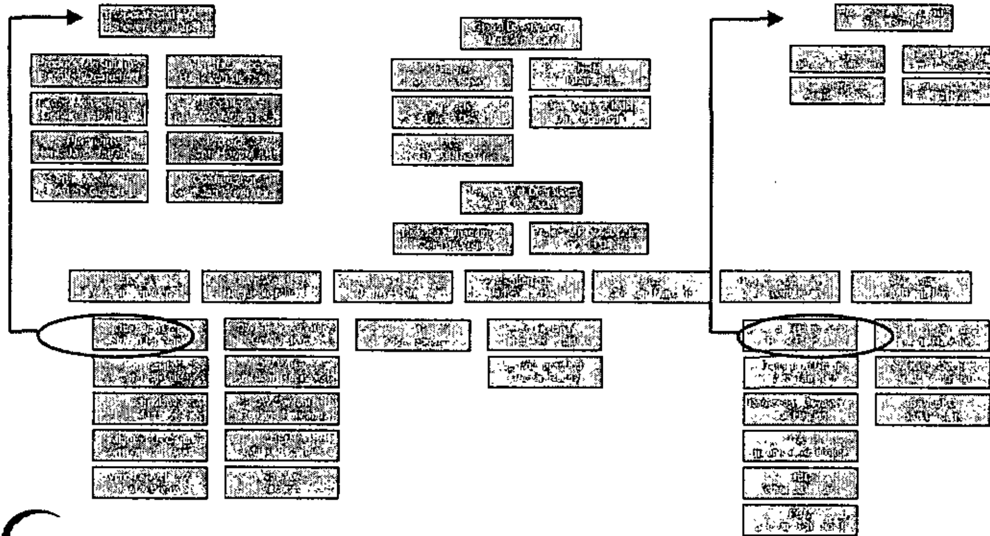
BAAN

Contents

- Testing in Baan Development
- Development of major releases
- Aspects around service packs
- Development of service packs
- Real life data & examples
- Pitfalls, tips & guidelines
- Wrap up.

invensys
Software Systems

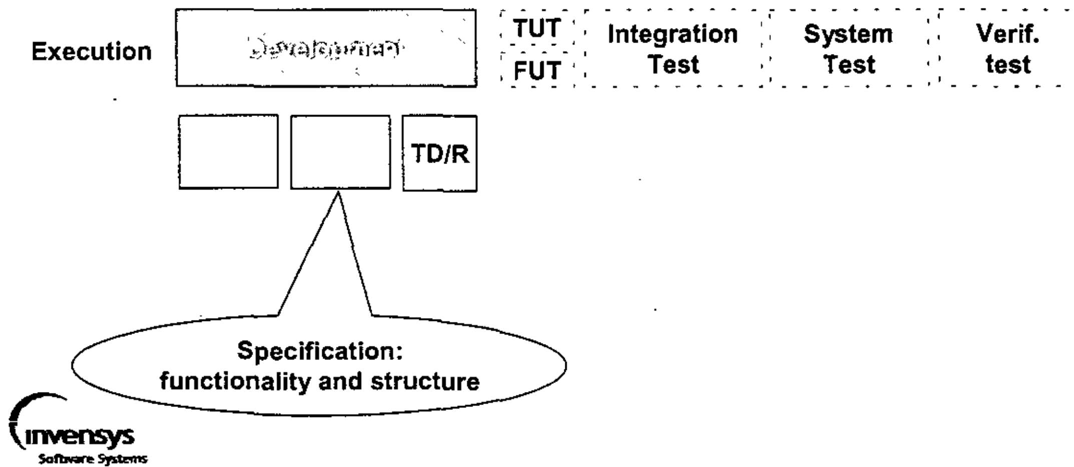
Testing in Baan Development



Development of major releases

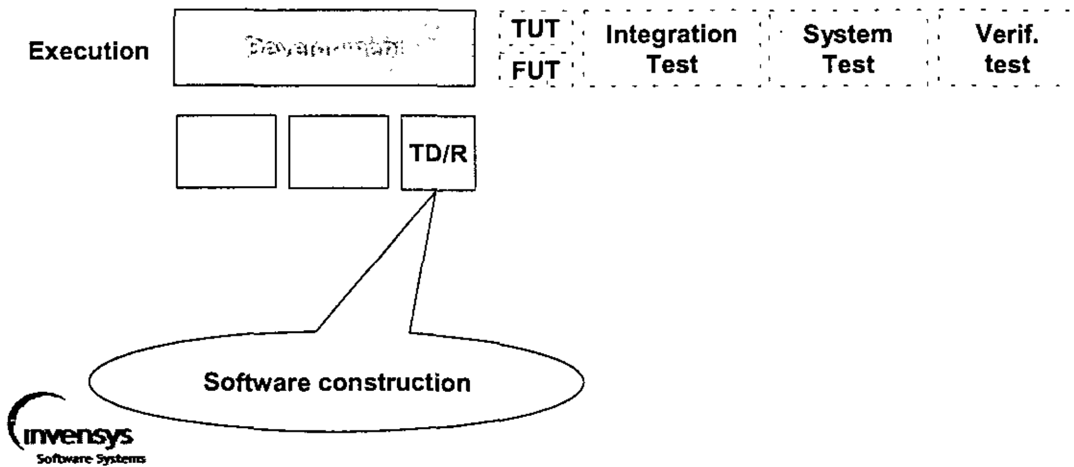
Development of major releases

BaaN



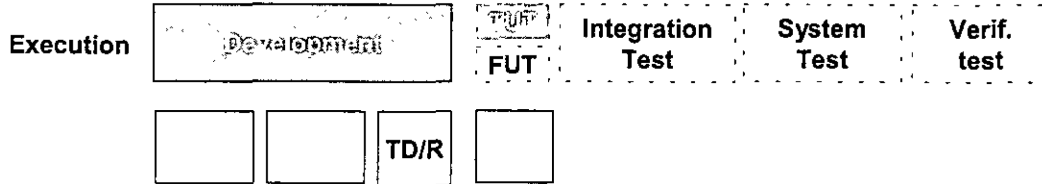
Development of major releases

BaaN



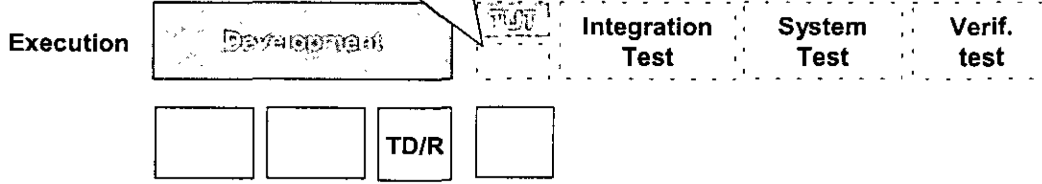
Development of major releases

Technical unit test (White box test)
knowledge of HOW it operates



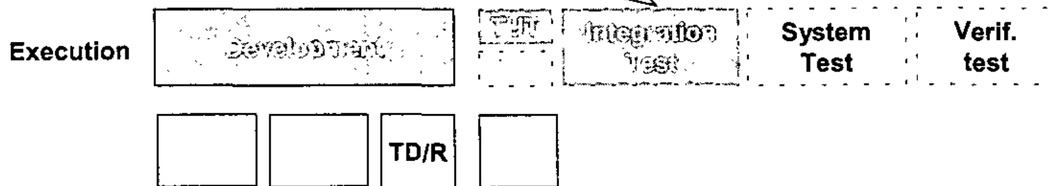
Development of major releases

Functional unit test (Black box test)
knowledge of WHAT it should do



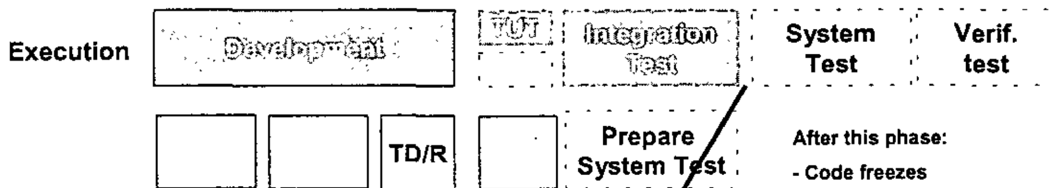
Development of major releases

Verification if all separate units do work correctly in conjunction with each other

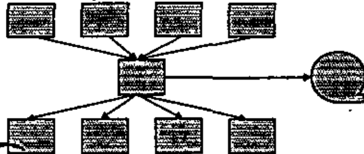


Development of major releases

Goal



Development environments



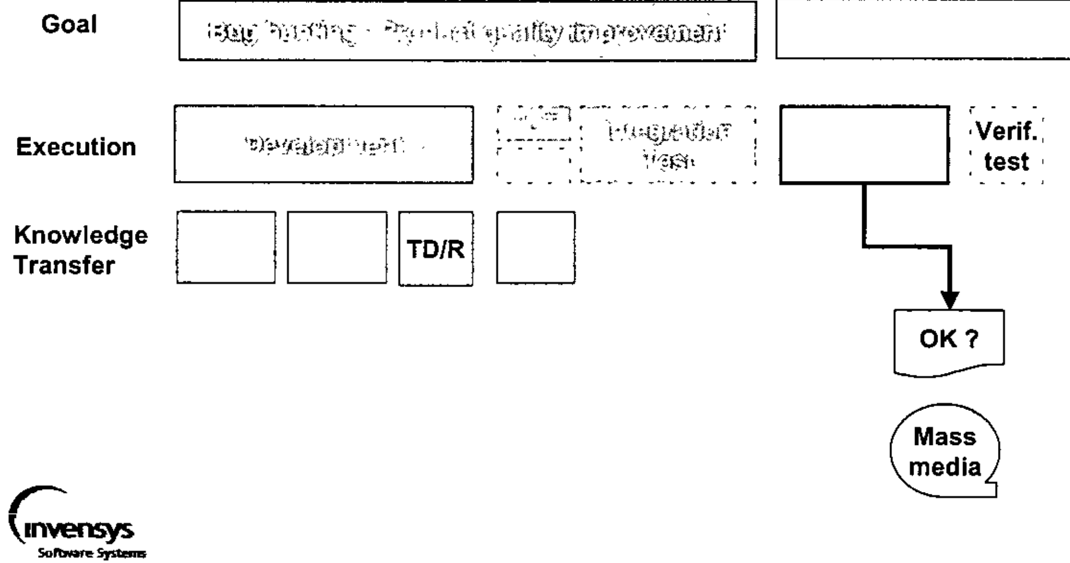
Verification environments

OK ?
Prep. ready?
Prod. ready?

After this phase:
- Code freezes
- independent systems
- release management
- formal software delivery
- usage of Install. Proced.
- etc

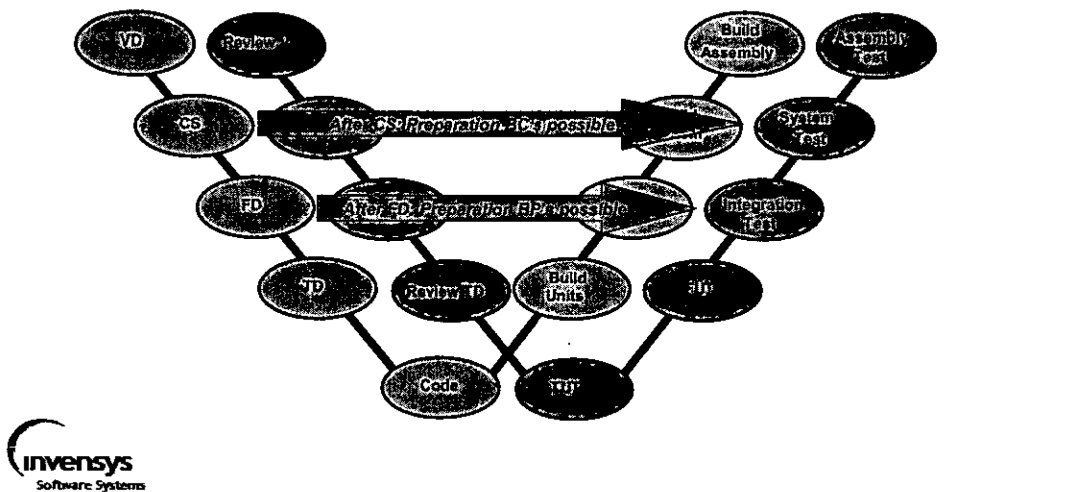
BaaN

Development of major releases



BaaN

Development of major releases



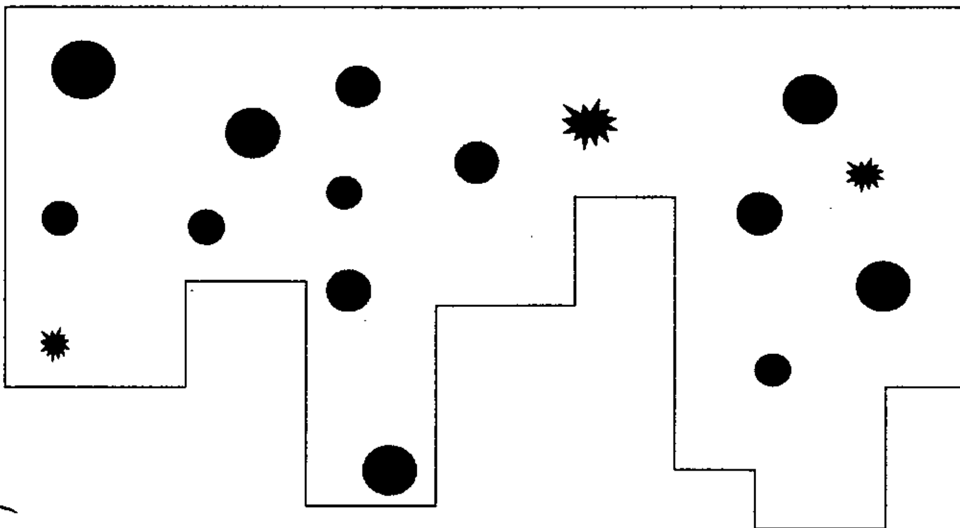
What are service packs?

Individual (requested) solutions - reactive

A collection of individual solutions/modifications delivered in a bundle to our customers - pro-active

(a solution can be triggered by a remark from the field or solutions can be a functional enhancement).

What are service packs?



Aspects around service packs

BaaN

- Less focus on functionality, more focus on quality -> more SP's
- Little modifications can affect the whole product
- Individual fixes can depend on each other
- Need for shorter delivery cycles (release train)
- When are service packs - good enough?
- Developers like to work on new development, but need to produce quality SP's
- Use of automated regression tests.

invensys Software Systems

Aspects around service packs

BaaN

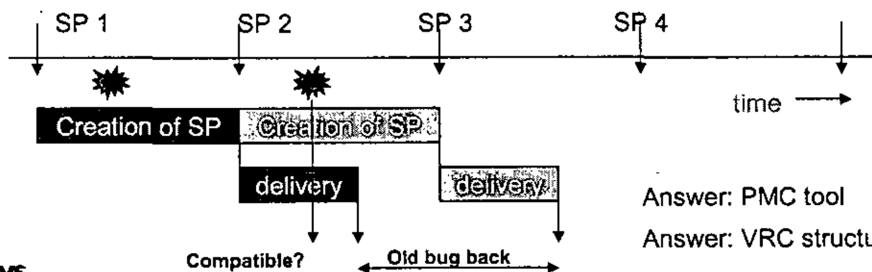
- Do we know the quality of our product after the installation of a Service Pack?
- Are customers happy with Service Packs?
- Example 1:
  - In Baan 5.0c SP2 we have 436 remarks solved
  - This resulted in 660 modified components
  - A customer asked for the solution of 10 problems
  - He received 436 - 10 = 426 fixes for free
  - But with a bad-fix rate of 5 % he gets also <u>21 new problems</u> for free.......

invensys Software Systems

Aspects around service packs

BAAN

- Do we know the quality of our product after the installation of a Service Pack?
- Are customers happy with Service Packs?
- Example 2:



Answer: PMC tool

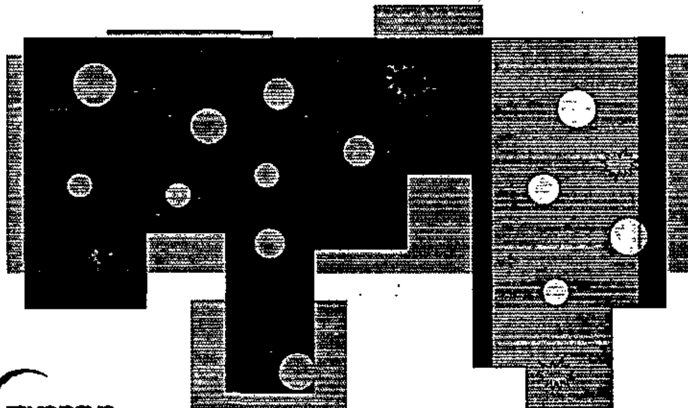
Answer: VRC structure

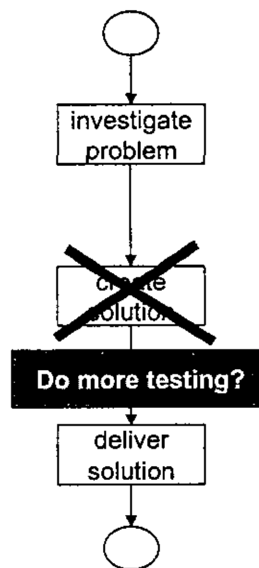
invensys Software Systems

Development of service packs

BAAN

What can we do to improve ?

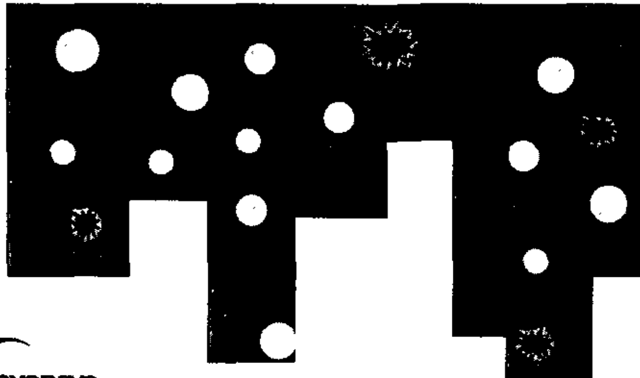


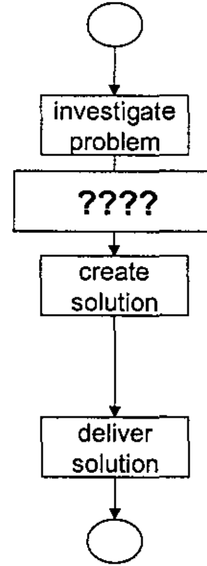


invensys Software Systems

Development of service packs

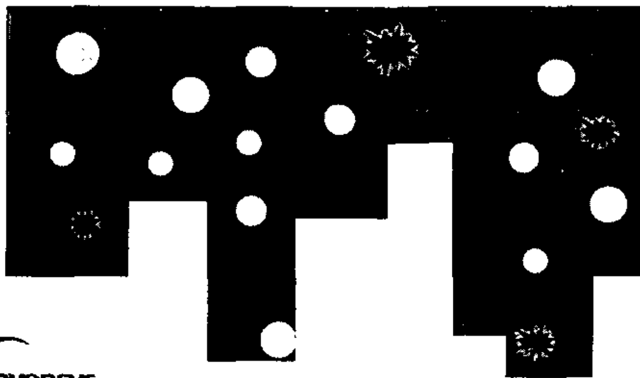
What can we do to improve ?

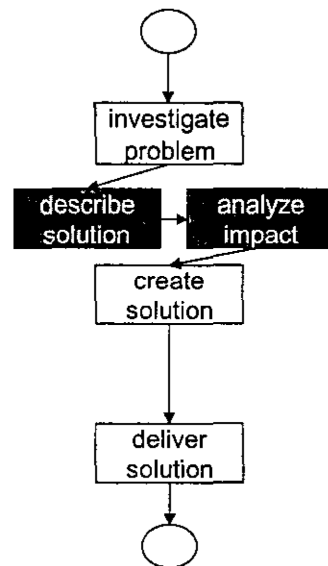




Development of service packs

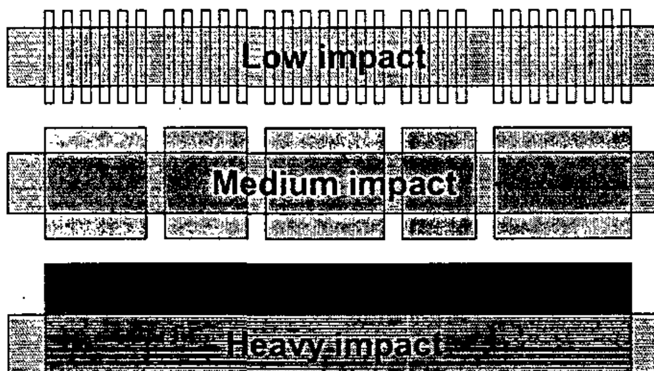
Analyze the impact of the change on the whole system

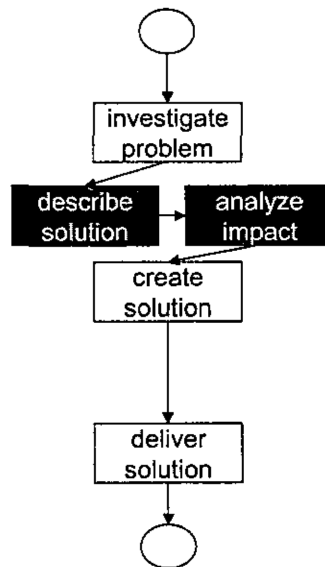




Development of service packs

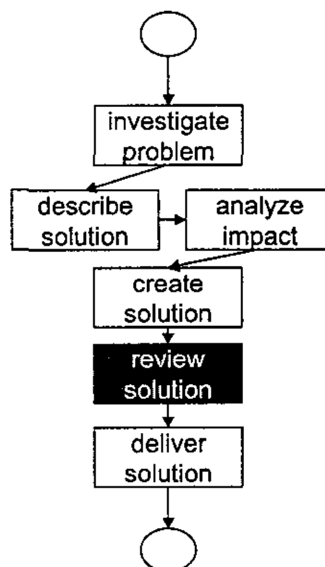
Categorize type of impact





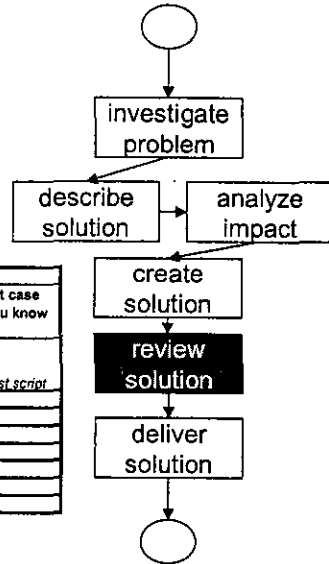
Development of service packs

Mandatory review of all software changes with a high impact by the owner of that software part



Development of service packs

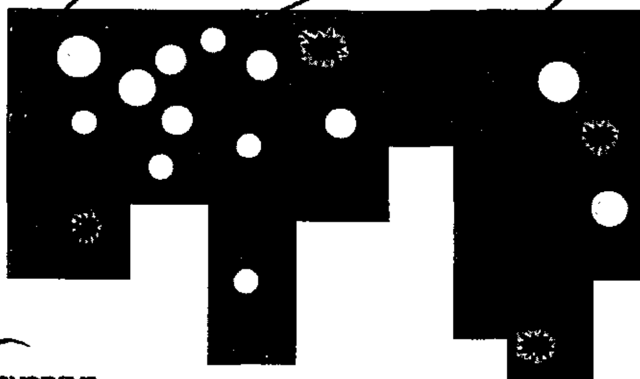
Results of the analysis <u>should</u> be registered

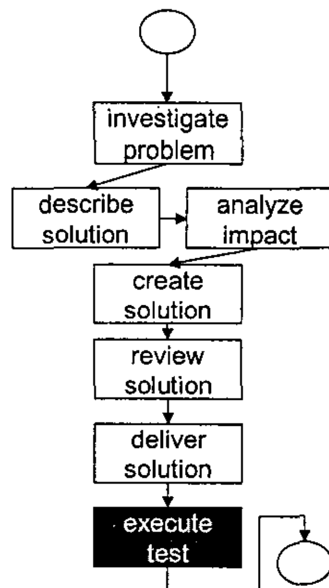


Development Group:		xxx				
Location:	**Remark ID:**	**Solution Reviewed by:**	**Severity of the Solution:**	**Additional Testing Required?**	**What should be tested?**	**Which test case (only if you know the case):**
[TCS field]	*[TCS field]*	*[TCS account]*	*[high/low]*	*[yes/no]*	*describe in words what the additional testing should be*	*refer to an existing test script*

Development of service packs

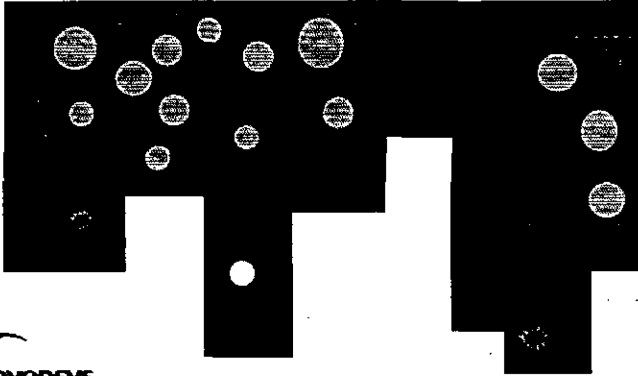
IST: re-testing

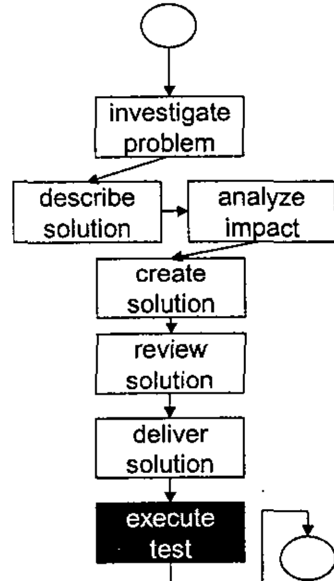




Development of service packs

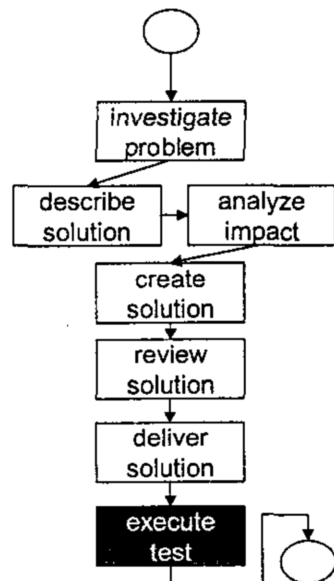
IAT: testing



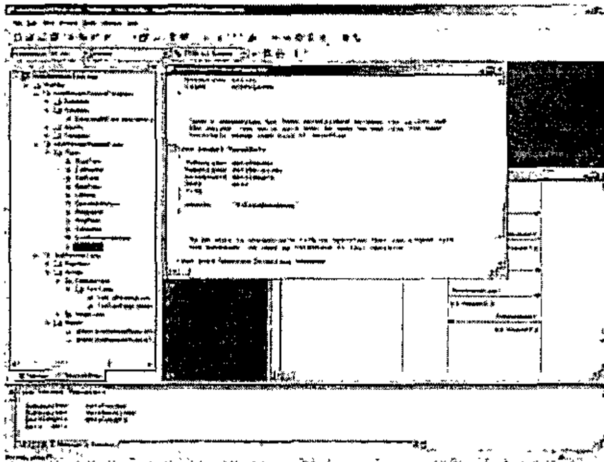


Development of service packs

ST: testing



TTCN – Testing into the future



An example of the graphical user interface of the new TTCN-3 test tool.

The testing part of the software development process is rapidly gaining in importance, and, with new technologies becoming more and more complex, there is an increased need for automated testing solutions to support both conformance to global standards and functional testing. Now there is a tool that meets the requirements.

Recently, Telelogic entered into an agreement with Nokia Research Center to develop a test tool based on the new globally standardized test language TTCN-3. The new test tool, which will be available commercially to all Telelogic customers, is designed for easy and rapid testing of systems and devices using a variety of technologies, including 3G, Internet protocols, distributed and heterogeneous systems.

"This joint development project," says Ingemar Ljungdahl, Chief Technology Officer at Telelogic, "confirms our strong market and technology position and also provides an example of how Telelogic is working with key customers in order to share their experience of using TTCN for advanced test engineering."

Testing in complex environments

"This is a very exciting time for the testing community," adds Richard Watson, Product Manager at Telelogic. "These new TTCN-3 tools see the birth of test development being performed in whichever form is most suitable for the user, for example UML, MSC, text, etc., while keeping costs down by being able to debug and execute all of these different 'views' on the same execution engine and equipment." According to Watson, "TTCN has provided the backbone for telecom conformance testing for the last ten years. The next generation, TTCN-3, marks the emergence of a testing architecture to support many other industries, including commerce, automotive, avionics and military."

Global standard

"TTCN-3 is a full review of the ISO 9646-3 test notation, TTCN-2," explains Johan Nordin, Telelogic's representative at the European Telecommunications Standards Institute (ETSI). "New areas of application testing are possible as TTCN-3 has been designed to address function/API-based testing as well as the event-based testing for which TTCN-2 is used today. TTCN-3 also includes features such as synchronous communication and dynamic parallel test configurations. Drawing on the experience and features of TTCN-2, the TTCN-3 core notation resembles a more conventional programming language in order to reduce the learning threshold," summarizes Nordin.

Easy to learn, easy to use

"Facilitating learning and making the jump between the different technologies as easily as possible were guiding principles for the tool design," adds Federico Engler, Senior Architect at Telelogic. "The backbone of the new testing tool has been developed using a generic Telelogic platform, internally called 'Telelogic Studio,' for development of a state-of-the-art user interface, which is also being used in other Telelogic products." With this common technology, it is possible to provide seamless tool integration and a coherent framework for the different Telelogic tool sets. "Because the graphical user interface is highly customizable, it can be tuned into virtually any desired user setup," concludes Engler.

Telelogic
infoNL@telelogic.com

Telelogic
Kaap Hoorndreef 30
3563 AT Utrecht
+31 30 265 1738

Real life data & examples

BaaN

																	Total
																	72
																	2078
																	5268
																	816
Total	6	1088	39	684	97	51	693	1000	700	411	1690	331	245	49	12	1138	8234

																	Total
																	21
																	139
																	851
																	186
Total		3	4	167	9	2	189	207	68	21	201	9	9	59	37	212	1197

Many changes on a component: inspection
Many changes in a solution: re testing

invensys
Software Systems

Real life data & examples

BaaN

- Individual solution test (IST) - done by development

																	Total
																	21
																	139
																	851
																	186
Total		3		167	9	2	189	207	68	21	201	9	0	59	37	21	1197

- Impacted Area Test (IAT) - on cp, ti, and wh

  tc covered by OCT
  td done in PSS test
  tools tested implicitly

- System Test (ST) - Duration two weeks - 5 full time testers, executing CFF in the first week, re-testing in the second week. (232 gr.md development - 5x5x2=50 gr.md testing - 1:5 ratio)

invensys
Software Systems

Pitfalls, tips & guidelines

BaaN

- Prevent loss of product ownership - act as co-builder
- Use lasting metrics that are easy to provide or produce
- Don't use a single quality indicator - use dashboard approach
- Plan evaluations as part of a deliverable
- Define objectives (e.g. FURPS) up-front
- Test-scope to be agreed with product owner (Q responsibility)
- Testing must be in the integrated project plan
- Use one single remark tracking and registration of hours.

invensys Software Systems

Wrap up

BaaN

- Testing is a part of development (product ownership!)
- Set objectives up-front and have them committed
- Metrics are crucial
- It is all about managing risks
- Delivery of SP's needs requires tooling and attention.

QUESTIONS ?

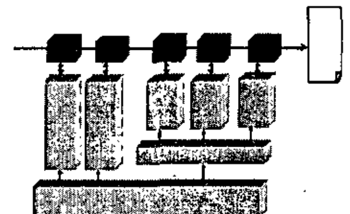
invensys Software Systems

Testing your Infrastructure

CMG
Information Technology

Testing your Infrastructure

Within many companies, local divisions and units invested in desktop hardware and software according to their own preferences. While this approach was sufficient at departmental or branch level, trying to exchange information across the company highlighted serious incompatibilities. Therefore, most of these organisations are now in the process of initiating company-wide desktop standardisation projects.



CMG gained experience in implementing so-called Test & Integration Centre's (TIC) for a number of large accounts. The main objective of a TIC is to define, develop and test the company-wide desktop standard as part of the whole infrastructure. In order to improve the quality, efficiency and effectiveness of the testing activities by testing the Infrastructure, CMG has often implemented TestFrame™ as an integral part of testing processes.

The advantage of using TestFrame™

When using TestFrame™ the tests are built in modules. A few advantages of using those modules are:

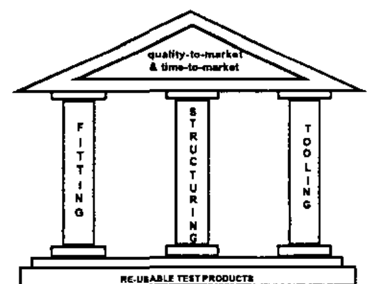
- *quality:* you can easily test the whole system (automated) instead of testing only the changes (regression and integration testing) without any human interaction (no human errors);
- *structuring*: by structuring the test in modules, you can use those modules to test any applications you want. So you can easily test many different desktops environments.
- *time saving*: you only have to define and automate the test (modules) once, the tests will then run automatically as often as necessary (sometimes even overnight).

TESTFRAME

Because CMG has great experience of testing Infrastructures with the help of TestFrame™, CMG has already developed a large number of standard tests (modules) for many standard office applications. These standard modules can be used by every small, medium or large organisation. You only have to develop tests for those applications that are specific to your organisation.

What is TestFrame™?

TestFrame™ is a methodology and the basic principles behind this methodology can be compared with the structure of a Greek temple. The roof symbolises your business objectives: high quality-to-market and short time-to-market. In order to achieve these objectives, we need a firm foundation, just as a temple needs a firm foundation. Within the CMG vision, this is based on the requirement that all testing material should be simple to maintain and should be reusable (modules). Three success factors act as the pillars supporting the roof: *fitting* (you can test everything, and it can be done in every organisation), *structuring* (by structuring your test, you get a higher quality and reusable products) and *tooling* (we can automate the testing by using any test-automation tool). The proper resources and the proper structure adapted to fit your organisation.



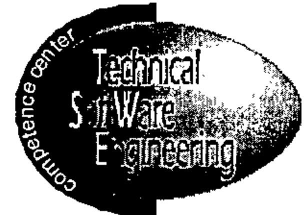
CMG Arnhem B.V.

Velperweg 37 PO-box 37 6801 HA Arnhem Tel +31-26- 3544544 Fax +31-26-3544566 info: hans.potze@cmg.nl (Associate Director)

ALKMAAR • AMSTELVEEN • AMSTERDAM • ARNHEM • DEN HAAG • EINDHOVEN • ENSCHEDE • GRONINGEN • MAASTRICHT • ROTTERDAM • UTRECHT
WOERDEN • BRUSSEL• FRANKFURT • HAMBURG • KEULEN • MÜNCHEN • STUTTGART • LONDON • MANCHESTER • PARIJS • NASHUA (USA) • SINGAPORE

ActiveLink
Technical Software Engineering

In embedded software development, communication between multiple processors is often required. Since many communication protocols (RS232, TCP/IP, JTAG, etc.) are available and proven standards for unified high-level communication are virtually absent, we developed *ActiveLink*; a small-sized tool for cross-platform communication.



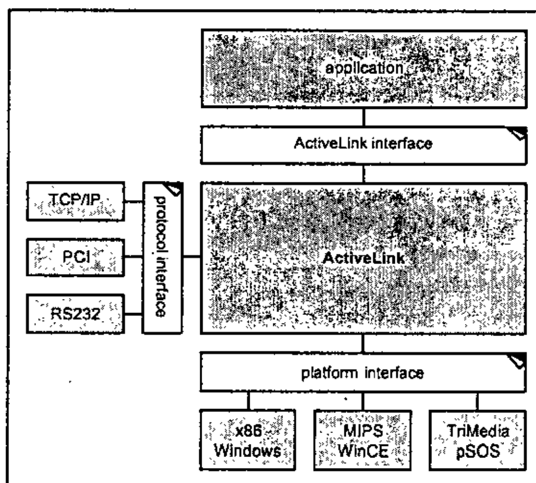
ActiveLink offers a Remote Procedure Call (RPC) mechanism as well as means to control remote memory, i.e., to allocate memory on a remote processor and to copy memory from a remote processor, and vice versa. ActiveLink enables the development of distributed applications in heterogeneous environments. It features:

- Transparent cross-platform communication by means of Remote Procedure Calls.
- Distributed memory space access.
- Provides a platform independent interface.
- Suited for embedded systems because of small memory footprint.
- Interfaces with programming languages C and C++.

Active Link

ActiveLink is available in two packages:
- The ActiveLink End-User Package supports the Windows and the MIPS/pSOS platform with TCP/IP and RS232 communication.
- The ActiveLink Developers Kit allows portability to any processor architecture (e.g., ARM, SPARC), operating system (e.g., WinCE, VxWorks, Linux), and communication protocol (e.g., PCI, USB).

TESTFRAME



ActiveLink architecture

We can offer:
- helpdesk support
- on site support and training by means of workshops
- support for porting ActiveLink to your platform
- an automated test environment with Embedded TestFrame
- support for development of distributed applications

ActiveLink has been integrated with *Embedded TestFrame*, the CMG solution for automated testing of embedded software.

CMG Eindhoven B.V.

For more information, please contact Gert-Jan van Dijk (gert-jan.van.dijk@cmg.nl) or Harro Jacobs (harro.jacobs@cmg.nl).

Luchthavenweg 57 PO Box 7089 5605 JB Eindhoven The Netherlands
Tel: +31-(0)40 – 29 57 777 Fax: +31-(0)40 – 29 57 630 Internet: www.cmg.nl

ALKMAAR • AMSTELVEEN • AMSTERDAM • ARNHEM • DEN HAAG • EINDHOVEN • ENSCHEDE • GRONINGEN • MAASTRICHT • ROTTERDAM • UTRECHT WOERDEN • BRUSSEL • FRANKFURT • HAMBURG • KEULEN • MÜNCHEN • STUTTGART • LONDON • MANCHESTER • PARIJS • NASHUA (USA) • SINGAPORE

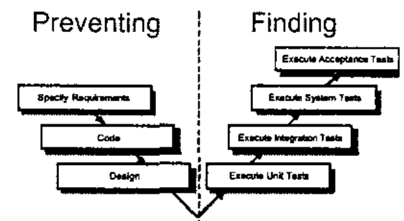
v2.0

Verification: Inspections and Testing

CMG
Information Technology

Verification is defined as confirming by examinations and provisions of objective evidence that specified requirements have been fulfilled. Efforts in verification are aimed at preventing and finding discrepancies (errors) between the requirements and the development deliverables.

Inspections are carried out in the activities on the left side of the Development V-model. Testing is carried out in the activities on the right side of the V-model.
A trade-off has to be made between the efforts devoted to prevention and to testing. This trade-off is necessary, since complete prevention of all errors and the finding of all errors are both impossible and would require unlimited resources. Practical experience has shown that the 80/20 rule applies in this trade-off: in the optimal situation, 80% of the errors should be found in inspections, whereas the remaining 20% should be found in the subsequent testing efforts.



Inspections
Inspections are an effective and efficient approach to prevent errors in the work products of a development project. Research and experiences have shown that inspections have a high return on investment: 50 errors found in inspections, save 250 hours in module testing.

Testing
A number of decisions need to be made when determining how to test the different work products:

- Primarily, an approach for testing should be determined; a choice should be made between random testing, or testing according to a recorded test plan including a strategy?
- Secondly, a method for testing should be determined; should tests be carried out ad-hoc, or should structured test specifications be developed that can be repeated?
- Finally, the choice can be made to execute the tests manually or to automate them.

TESTFRAME

ActiveLink

CMG can offer

- Workshop, training and implementation of inpections
- Implement a method of structured testing , i.e. TestFrame
- Automate testing using the TestFrame method

CMG
CMG Eindhoven B.V.

For more information, you can contact Roland Kieboom (Assistant Director) at Roland.kieboom@cmg.nl.

Luchthavenweg 57 Postbus 7089 5605 JB Eindhoven
Tel: 040 – 29 57 777 Fax: 040 – 29 57 630 Internet: www.cmg.nl

ALKMAAR • AMSTELVEEN • AMSTERDAM • ARNHEM • DEN HAAG • EINDHOVEN • ENSCHEDE • GRONINGEN • MAASTRICHT • ROTTERDAM • UTRECHT
WOERDEN • BRUSSEL • FRANKFURT • HAMBURG • KEULEN • MÜNCHEN • STUTTGART • LONDON • MANCHESTER • PARIJS • NASHUA (USA) • SINGAPORE

vs 1.0

Training
Software Inspectie

CMG
Information Technology

Sofware Inspectie is een methode waarmee vroegtijdig in het software ontwikkelproces op een effectieve en efficiënte wijze issues, die uiteindelijk zouden leiden tot fouten in het eindproduct, kunnen worden opgespoord. Het doel is om het aantal fouten in het eindproduct tot een minimum te reduceren en de kosten voor het vinden van deze fouten te beperken. Software inspecties kunnen worden toegepast voor zowel ontwikkeldocumentatie en source code.

Inhoud van de training:

- **Vergelijking van inspecties met reviews, walkthroughs etc.**: verschillen tussen een inspectie, review en een walkthrough en het toepassen van deze methodes
- **Inspectie principes:** basis principes van inspecties, korte introductie van wat inspecties nu eigenlijk zijn
- **Inspectie process:** stap voor stap doorlopen van de 8 fasen van het inspectie proces: planning; kick-off; checking, logging, process brainstorming, edit; follow-up en exit
- **Entry/Exit criteria:** wat zijn de criteria om met een inspectie te kunnen starten en wat zijn de criteria om de inspectie te beëindigen en het geïnspecteerde product goed te keuren
- **Sampling methode:** het vergroten van de efficiëntie van het inspectie proces door verschillende personen delen (samples) van het product te laten inspecteren
- **Metrics:** het gebruik van metrics om de efficiëntie en de effectiviteit van de inspectie te vergroten
- **Checklists:** het gebruik van checklists om de kans op het vinden van major issues te vergroten
- **workshops:** het uitvoeren van inspecties op eigen documenten met als doel het in de praktijk leren van het proces en het leren vinden van major issues.

Doelgroep
Software engineers, testers, kwaliteits medewerkers en andere medewerkers die betrokken (zullen) zijn bij de inspectie van documenten en/of code.

Instapprofiel
2 Jaar kennis/ervaring om de producten (documenten/code) die voor inspectie worden aangeboden te kunnen beoordelen.

Voorbereiding
De deelnemers dienen een document of listings van source code mee te brengen waarmee zij tijdens de workshops het inspectie proces kunnen trainen.

Aantal dagen
De cursus omvang is 2 opeenvolgende dagen.

Rendement van de training
Ieder major fout die tijdens de inspectie wordt gevonden levert een besparing van gemiddeld 8 uur op voor het totale project waarin de kosten voor de inspectie zijn inbegrepen.

Voor meer informatie over de mogelijkheden voor een groepstraining bij u in het bedrijf kunt u vrijblijvend contact opnemen met uw CMG accountmanager of het SPI competence centre in Eindhoven.

CMG is member of

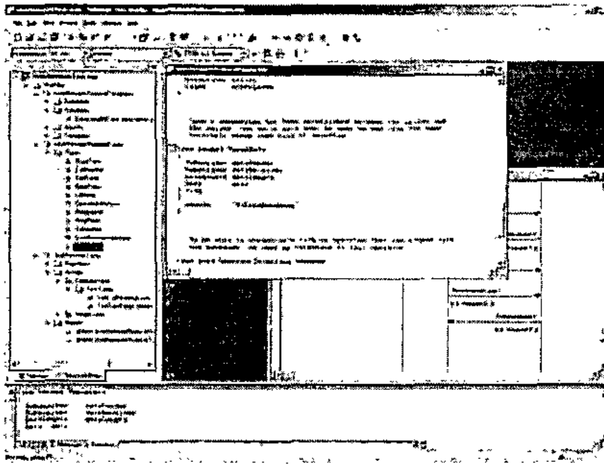
ESI European Software Institute

CMG
CMG Eindhoven B.V.

Luchthavenweg 57 Postbus 7089 5605 JB Eindhoven Tel: 040 -29 57 777 Fax: 040 – 29 57 630 www.cmg.nl/spi

ALKMAAR • AMSTELVEEN • AMSTERDAM • ARNHEM • DEN HAAG • EINDHOVEN • ENSCHEDE • GRONINGEN • MAASTRICHT • ROTTERDAM • UTRECHT WOERDEN • BRUSSEL• FRANKFURT • HAMBURG • KEULEN • MÜNCHEN • STUTTGART • LONDON • MANCHESTER • PARIJS • NASHUA (USA) • SINGAPORE

TTCN – Testing into the future



An example of the graphical user interface of the new TTCN-3 test tool.

The testing part of the software development process is rapidly gaining in importance, and, with new technologies becoming more and more complex, there is an increased need for automated testing solutions to support both conformance to global standards and functional testing. Now there is a tool that meets the requirements.

Recently, Telelogic entered into an agreement with Nokia Research Center to develop a test tool based on the new globally standardized test language TTCN-3. The new test tool, which will be available commercially to all Telelogic customers, is designed for easy and rapid testing of systems and devices using a variety of technologies, including 3G, Internet protocols, distributed and heterogeneous systems.

"This joint development project," says Ingemar Ljungdahl, Chief Technology Officer at Telelogic, "confirms our strong market and technology position and also provides an example of how Telelogic is working with key customers in order to share their experience of using TTCN for advanced test engineering."

Testing in complex environments

"This is a very exciting time for the testing community," adds Richard Watson, Product Manager at Telelogic. "These new TTCN-3 tools see the birth of test development being performed in whichever form is most suitable for the user, for example UML, MSC, text, etc., while keeping costs down by being able to debug and execute all of these different 'views' on the same execution engine and equipment." According to Watson, "TTCN has provided the backbone for telecom conformance testing for the last ten years. The next generation, TTCN-3, marks the emergence of a testing architecture to support many other industries, including commerce, automotive, avionics and military."

Global standard

"TTCN-3 is a full review of the ISO 9646-3 test notation, TTCN-2," explains Johan Nordin, Telelogic's representative at the European Telecommunications Standards Institute (ETSI). "New areas of application testing are possible as TTCN-3 has been designed to address function/API-based testing as well as the event-based testing for which TTCN-2 is used today. TTCN-3 also includes features such as synchronous communication and dynamic parallel test configurations. Drawing on the experience and features of TTCN-2, the TTCN-3 core notation resembles a more conventional programming language in order to reduce the learning threshold," summarizes Nordin.

Easy to learn, easy to use

"Facilitating learning and making the jump between the different technologies as easily as possible were guiding principles for the tool design," adds Federico Engler, Senior Architect at Telelogic. "The backbone of the new testing tool has been developed using a generic Telelogic platform, internally called 'Telelogic Studio,' for development of a state-of-the-art user interface, which is also being used in other Telelogic products." With this common technology, it is possible to provide seamless tool integration and a coherent framework for the different Telelogic tool sets. "Because the graphical user interface is highly customizable, it can be tuned into virtually any desired user setup," concludes Engler.

Telelogic
infoNL@telelogic.com

Telelogic
Kaap Hoorndreef 30
3563 AT Utrecht
+31 30 265 1738

Computer Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: m.m.j.l.philips@tue.nl (we cannot guarantee the availability of the requested reports)

In this series appeared:

97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.
97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.
97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28
97/15	M. Franssen	λP-: A Pure Type System for First Order Loginc with Automated Theorem Proving, p.35.
97/16	W.M.P. van der Aalst	On the verification of Inter-organizational workflows, p. 23
97/17	M. Vaccari and R.C. Backhouse	Calculating a Round-Robin Scheduler, p. 23.
97/18	Werkgemeenschap Informatiewetenschap redactie: P.M.E. De Bra	Informatiewetenschap 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie Informatiewetenschap, p. 60.
98/01	W. Van der Aalst	Formalization and Verification of Event-driven Process Chains, p. 26.
98/02	M. Voorhoeve	State / Event Net Equivalence, p. 25
98/03	J.C.M. Baeten and J.A. Bergstra	Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04	R.C. Backhouse	Pair Algebras and Galois Connections, p. 14
98/05	D. Dams	Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06	G. v.d. Bergen, A. Kaldewaij V.J. Dielissen	Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07	Proceedings of the workhop on Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98) June 22, 1998 Lisbon, Portugal	edited by W. v.d. Aalst, p. 209
98/08	Informal proceedings of the Workshop on User Interfaces for Theorem Provers. Eindhoven University of Technology ,13-15 July 1998	edited by R.C. Backhouse, p. 180

98/09	K.M. van Hee and H.A. Reijers	An analytical method for assessing business processes, p. 29.
98/10	T. Basten and J. Hooman	Process Algebra in PVS
98/11	J. Zwanenburg	The Proof-assistemt Yarrow, p. 15
98/12	Ninth ACM Conference on Hypertext and Hypermedia Hypertext '98 Pittsburgh, USA, June 20-24, 1998 Proceedings of the second workshop on Adaptive Hypertext and Hypermedia.	Edited by P. Brusilovsky and P. De Bra, p. 95.
98/13	J.F. Groote, F. Monin and J. v.d. Pol	Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.
99/01	V. Bos and J.J.T. Kleijn	Structured Operational Semantics of χ, p. 27
99/02	H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst	Diagnosing Workflow Processes using Woflan, p. 44
99/03	R.C. Backhouse and P. Hoogendijk	Final Dialgebras: From Categories to Allegories, p. 26
99/04	S. Andova	Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81
99/05	M. Franssen, R.C. Veltkamp and W. Wesselink	Efficient Evaluation of Triangular B-splines, p. 13
99/06 66	T. Basten and W. v.d. Aalst	Inheritance of Workflows: An Approach to tackling problems related to change, p.
99/07	P. Brusilovsky and P. De Bra	Second Workshop on Adaptive Systems and User Modeling on the World Wide Web, p. 119.
99/08 VFM'99	D. Bosnacki, S. Mauw, and T. Willemse	Proceedings of the first international syposium on Visual Formal Methods -
99/09	J. v.d. Pol, J. Hooman and E. de Jong	Requirements Specification and Analysis of Command and Control Systems
99/10	T.A.C. Willemse	The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μCRL, p. 44.
99/11	J.C.M. Baeten and C.A. Middelburg	Process Algebra with Timing: Real Time and Discrete Time, p. 50.
99/12	S. Andova	Process Algebra with Probabilistic Choice, p. 38.
99/13	K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen	A Framework for Component Based Software Architectures, p. 19
99/14	A. Engels and S. Mauw	Why men (and octopuses) cannot juggle a four ball cascade, p. 10
99/15	J.F. Groote, W.H. Hesselink, S. Mauw, R. Vermeulen	An algorithm for the asynchronous *Write-All* problem based on process collision*, p. 11.
99/16	G.J. Houben, P. Lemmens	A Software Architecture for Generating Hypermedia Applications for Ad-Hoc Database Output, p. 13.
99/17	T. Basten, W.M.P. v.d. Aalst	Inheritance of Behavior, p.83
99/18	J.C.M. Baeten and T. Basten	Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79
99/19	J.C.M. Baeten and C.A. Middelburg	Real Time Process Algebra with Time-dependent Conditions, p.33.
99/20	Proceedings Conferentie Informatiewetenschap 1999 Centrum voor Wiskunde en Informatica 12 november 1999, p.98	edited by P. de Bra and L. Hardman
00/01	J.C.M. Baeten and J.A. Bergstra	Mode Transfer in process Algebra, p. 14
00/02	J.C.M. Baeten	Process Algebra with Explicit Termination, p. 17.
00/03	S. Mauw and M.A. Reniers	A process algebra for interworkings, p. 63.
00/04	R. Bloo, J. Hooman and E. de Jong	Semantical Aspects of an Architecture for Distributed Embedded Systems*, p. 47.
00/05	J.F. Groote and M.A. Reniers	Algebraic Process Verification, p. 65.

00/06	J.F. Groote and J. v. Wamel	The Parallel Composition of Uniform Processes wit Data, p. 19
00/07	C.A. Middelburg	Variable Binding Operators in Transition System Specifications, p. 27.
00/08	I.D. van den Ende	Grammars Compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases, p. 33.
00/09	R.R. Hoogerwoord	A Formal Development of Distributed Summation, p. 35
00/10	T. Willemse, J. Tretmans and A. Klomp	A Case Study in Formal Methods: Specification and Validation on the OM/RR Protocol, p. 14.
00/11	T. Basten and D. Bošnački	Enhancing Partial-Order Reduction via Process Clustering, p. 14
00/12	S. Mauw, M.A. Reniers and T.A.C. Willemse	Message Sequence Charts in the Software Engineering Process, p. 26
00/13	J.C.M. Baeten, M.A. Reniers	Termination in Timed Process Algebra, p. 36
00/14	M. Voorhoeve, S. Mauw	Impossible Futures and Determinism, p. 14
00/15	M. Oostdijk	An Interactive Viewer for Mathematical Content based on Type Theory, p. 24.
00/16	F. Kamareddine, R. Bloo, R. Nederpelt	Characterizing λ-terms with equal reduction behavior, p. 12
00/17	T. Borghuis, R. Nederpelt	Belief Revision with Explicit Justifications: an Exploration in Type Theory, p. 30.
00/18	T. Laan, R. Bloo, F. Kamareddine, R. Nederpelt	Parameters in Pure Type Systems, p. 41.
00/19	J. Baeten, H. van Beek, S. Mauw	Specifying Internet applications with *DiCons*, p. 9
00/20	Editors: P. v.d. Vet and P. de Bra	Proceedings: Conferentie Informatiewetenschap 2000, De Doelen, Utrecht, 5 april 2000, p. 98
01/01	H. Zantema and J. v.d. Pol	A Rewriting Approach to Binary Decision Diagrams, p. 27
01/02	T.A.C. Willemse	Interpretations of Automata, p. 41
01/03	G.I. Joigov	Systems for Open Terms: An Overview, p. 39
01/04	P.J.L. Cuijpers, M.A. Reniers, A.G. Engels	Beyond Zeno-behaviour, p. 15
01/05	D.K. Hammer, J. Hooman, M.A. Reniers, O. van Roosmalen, A. Sintotski	Design of the mine pump control system, p. 69
01/06	J.C.M. Baeten and E.P. de Vink	Axiomatizing GSOS with termination, p. 22
01/10	7 e Nederlandse Testdag editors L.M.G. Feijs, N. Goga, S. Mauw, T.A.C. Willemse	

TU/e technische universiteit eindhoven

P.O. Box 513
5600 MB Eindhoven
The Netherlands

/ department of mathematics and computing science