Proceedings of VVSS2007 - verification and validation of software systems, 23rd March 2007, Eindhoven, The Netherlands

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Proceedings of

VVSS2007 – Verification and Validation of Software Systems

23^{rd} March 2007, Eindhoven, the Netherlands

Editors:
Perry Groot
Alexander Serebrenik
Marko van Eekelen

Organised by LaQuSo – Laboratory for Quality Software

TUE Computer Science Reports 07-04
ISSN 0926–4515

Series editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Table of Content

Preface

Keynote and Speaker Presentations

Track 1 - Requirements 1 - Track chair: Hans van Vliet

Track 2 - Performance - Track chair: Johan Lukkien

Track 3 - Embedded 1 - Track chair: Ed Brinksma

Track 4 - New Trends in Testing 1 - Track chair: Pieter Koopman

Track 5 - Models 1 - Track chair: Jos Baeten

Track 6 - Requirements 2 - Track chair: Jan Dietz

Track 7 - New Trends in Testing 2 - Track chair: Jan Tretmans

Track 8 - Embedded 2 - Track chair: Arend Rensink

Track 9 - Quality Checking - Track chair: Jos Trienekens

Track 10 - Models 2 - Track chair: Jan Friso Groote

Track 11 - Open Source - Track chair: Yaroslav Usenko

Track 12 - New Trends in Testing 3 - Track chair: Judi Romijn

Track 13 - Embedded 3 - Track chair: Roelof Hamberg

Track 14 - Measuring Quality - Track chair: Marko van Eekelen

Track 15 - Security - Track chair: Bart Jacobs

Preface

VVSS 2007 (Verification and Validation of Software Systems) is the third symposium annex tool exhibition that is launched by LaQuSo (Laboratory for Quality Software) to exchange experiences about methods and techniques among decision makers and experts in the domains of software testing, quality assurance and formal methods. This year VVSS took place in Eindhoven, the Netherlands, on March 23, 2007.

This volume contains slides, paper, or abstract of the presentations given at Eindhoven on March 23, 2007. The program this year includes two keynote speakers: Prof. Dr. W.M.P. van der Aalst (Eindhoven University of Technology, the Netherlands) and Prof. Dr. D.L. Parnas (University of Limerick, Ireland). The technical part of the program was provided by thirty industrial and academic speakers from Belgium, Germany, the Netherlands, Poland, and the United Kingdom. Following the tradition of the previous VVSS meetings we were happy to welcome tool exhibitioners and poster presenters.

Organizing the symposium would have been impossible without the support of Willeke Quaedflieg, Mark van den Brand, Perry Groot, Geert Kemps, and Henk Schimmel. We also would like to thank the track chairs, LaQuSo members and LaQuSo program board for their contribution.

VVSS 2007 Programme Chairs:

Alexander Serebrenik
Eindhoven University of Technology
LaQuSo Eindhoven

and

Marko van Eekelen
Radboud University Nijmegen
LaQuSo Nijmegen

Posters

- Ed Brandt (Refis, the Netherlands), Alessandro Di Bucchianico (Eindhoven University of Technology, the Netherlands). “A new statistical tool for supporting software testing”.
- Ed Brandt (Refis, the Netherlands), Alessandro Di Bucchianico (Eindhoven University of Technology, the Netherlands). “Working group test metrics”.
- Pieter Claassen, Eric Verheul (Radboud University Nijmegen and Pricewaterhouse-Coopers, the Netherlands). “Browse Risk frOm unWarranted Security Exceptions (BROWSE)”.
- Francois Degrave (University of Namur, Belgium), Nathalie Mweze (University of Namur), Badouin Lecharlier (Universit Catholique de Louvain, Belgium), Wim Vanhoof (University of Namur, Belgium). “Automatic generation of test inputs for Mercury programs”.
- Perry Groot, Marko van Eekelen, Arjen Hommersom, Peter Lucas, Alexander Serebrenik, Yaroslav Usenko, Hajo Reijers (LaQuSo, the Netherlands). “Medical Guidelines - Past, Present, and Future”.
- Matthijs Mekking (Radboud University Nijmegen, the Netherlands). “A Proposed Internet Standard in UPPAAL”.
- Gregor Panovski (Eindhoven University of Technology, the Netherlands). “Quality Assessment of Product Software”.
- Wilco Schumacher (Collis, the Netherlands). “Added value of a conceptual model for performance testing”.
- David Van Bedaf, Anne Kerckx, Lien Keulemans, Alex Van Cauwenbergh (Quasus, Belgium). “Validation in a Paperless World: a real life example”.
- Martijn Visscher (Logica CMG, the Netherlands). “Successful Testmanagement : a 360° Solution”.
- Chris George (United Nations University / International Institute for Software, Macao). “RAISE tools from UNU/IIST”.

Tool exhibitors

- AbsInt Angewandte Informatik GmbH (Germany)
- Atos Origin Nederland B.V. (The Netherlands)
- Borland B.V. (The Netherlands)
- Collis B.V. (The Netherlands)
- Compuware B.V. (The Netherlands)
- Coverity, Inc. (USA)
- IFSQ, Institute for Software Quality (The Netherlands)
- Imtech ICT Technical Systems (The Netherlands)
- LDRA Ltd./INDES - Integrated Development Solutions B.V. (The Netherlands)
- Mithun Training & Consulting B.V. (The Netherlands)
- Parasoft Netherlands B.V. (The Netherlands)
- Programming Research B.V. (The Netherlands)
- ps_testware B.V. (The Netherlands)
- QSM-Europe B.V. (The Netherlands)
- Rescop (The Netherlands)
- SOMS Software Tools (The Netherlands)
- Telelogic Netherlands B.V. (The Netherlands)
- Verifysoft Technology GmbH (Germany)

VVSS 2007

Keynote and Speaker Presentations

ProM 4.0: Comprehensive Support for *Real* Process Analysis

W.M.P. van der Aalst[1], B.F. van Dongen[1], C.W. Günther[1], R.S. Mans[1], A.K. Alves de Medeiros[1], A. Rozinat[1], V. Rubin[2,1], M. Song[1], H.M.W. Verbeek[1], and A.J.M.M. Weijters[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
{w.m.p.v.d.aalst}@tue.nl
[2] University of Paderborn, Paderborn, Germany

Abstract. This tool paper describes the functionality of *ProM*. Version 4.0 of ProM has been released at the end of 2006 and this version reflects recent achievements in *process mining*. Process mining techniques attempt to extract non-trivial and useful information from so-called "event logs". One element of process mining is *control-flow discovery*, i.e., automatically constructing a process model (e.g., a Petri net) describing the causal dependencies between activities. Control-flow discovery is an interesting and practically relevant challenge for Petri-net researchers and ProM provides an excellent platform for this. For example, the theory of regions, genetic algorithms, free-choice-net properties, etc. can be exploited to derive Petri nets based on example behavior. However, as we will show in this paper, the functionality of ProM 4.0 is not limited to control-flow discovery. ProM 4.0 also allows for the discovery of other perspectives (e.g., data and resources) and supports related techniques such as conformance checking, model extension, model transformation, verification, etc. This makes ProM a versatile tool for process analysis which is not restricted to model analysis but also includes log-based analysis.

1 Introduction

The first version of ProM was released in 2004. The initial goal of ProM was to unify process mining efforts at Eindhoven University of Technology and other cooperating groups [4]. Traditionally, most analysis tools focusing on processes are restricted to *model-based analysis*, i.e., a model is used as the starting point of analysis. For example, the alternating-bit protocol can be modeled as a Petri net and verification techniques can then be used to check the correctness of the protocol while simulation can be used to estimate performance aspects. Such analysis is only useful *if the model reflects reality.* Process mining techniques use *event logs* as input, i.e., information recorded by systems ranging from information systems to embedded systems. Hence the starting point is not a model but the observed reality. Therefore, we use the phrase *real process analysis* to position process mining with respect to classical model-based analysis. Note that

ProM also uses models (e.g., Petri nets). However, these models (1) are discovered from event logs, (2) are used to reflect on the observed reality (conformance checking), or (3) are extended based on information extracted from logs.

Process mining is relevant since more and more information about processes is collected in the form of event logs. The widespread use of information systems, e.g., systems constructed using ERP, WFM, CRM, SCM, and PDM software, resulted in the omnipresence of vast amounts of event data. Events may be recorded in the form of audit trails, transactions logs, or databases and may refer to patient treatments, order processing, claims handling, trading, travel booking, etc. Moreover, recently, more and more devices started to collect data using TCP/IP, GSM, Bluetooth, and RFID technology (cf. high-end copiers, wireless sensor networks, medical systems, etc.).

Table 1. Comparing ProM 1.1 presented in [7] with ProM 4.0.

Version	ProM 1.1	ProM 4.0
Mining plug-ins	6	27
Analysis plug-ins	7	35
Import plug-ins	4	16
Export plug-ins	9	28
Conversion plug-ins	3	22
Log filter plug-ins	0	14
Total number of plug-ins	29	142

At the Petri net conference in 2005, Version 1.1 of ProM was presented [7]. In the last two years ProM has been extended dramatically and currently dozens of researchers are developing plug-ins for ProM. ProM is open source and uses a plug-able architecture, e.g., people can add new process mining techniques by adding plug-ins without spending any efforts on the loading and filtering of event logs and the visualization of the resulting models. An example is the plug-in implementing the α-algorithm [5], i.e., a technique to automatically derive Petri nets from event logs. The version of ProM presented at the Petri net conference in 2005 (Version 1.1) contained only 29 plug-ins. Version 4.0 provides 142 plug-ins, i.e., there are almost five times as many plug-ins. Moreover, there have been spectacular improvements in the quality of mining algorithms and the scope of ProM has been extended considerably. This is illustrated by Table 1 which compares the version presented in [7] with the current version. To facilitate the understanding of Table 1, we briefly describe the six types of plug-ins:

- *Mining plug-ins* implement some mining algorithm, e.g., the α-miner to discover a Petri net [5] or the social network miner to discover a social network [1].
- *Export plug-ins* implement some “save as” functionality for specific objects in ProM. For example, there are plug-ins to save Petri nets, EPCs, social networks, YAWL, spreadsheets, etc. often also in different formats (PNML, CPN Tools, EPML, AML, etc.).
- *Import plug-ins* implement an “open” functionality for specific objects, e.g., load instance-EPCs from ARIS PPM or BPEL models from WebSphere.

– *Analysis plug-ins* which typically implement some property analysis on some mining result. For example, for Petri nets there is a plug-in which constructs place invariants, transition invariants, and a coverability graph. However, there are also analysis plug-ins to compare a log and a model (i.e., conformance checking) or a log and an LTL formula. Moreover, there are analysis plug-ins related to performance measurement (e.g., projecting waiting times onto a Petri net).
– *Conversion plug-ins* implement conversions between different data formats, e.g., from EPCs to Petri nets or from Petri nets to BPEL.
– *Log filter plug-ins* implement different ways of "massaging" the log before applying process mining techniques. For example, there are plug-ins to select different parts of the log, to abstract from infrequent behavior, clean the log by removing incomplete cases, etc.

In this paper we do not elaborate on the architecture and implementation framework for plug-ins (for this we refer to [7]). Instead we focus on the functionality provided by the many new plug-ins in ProM 4.0.

The remainder of this paper is organized as follows. Section 2 provides an overview of process mining and briefly introduces the basic concepts. Section 3 describes the "teleclaims" process of an Australian insurance company. A log of this process is used as a running example and is used to explain the different types of process mining: Discovery (Section 4), Conformance (Section 5), and Extension (Section 6). Section 7 briefly mentions additional functionality such as verification and model transformation. Section 8 concludes the paper.

2 Overview

The idea of process mining is to discover, monitor and improve *real* processes (i.e., not assumed processes) by extracting knowledge from event logs. Today many of the activities occurring in processes are either supported or monitored by information systems. Consider for example ERP, WFM, CRM, SCM, and PDM systems to support a wide variety of business processes while recording well-structured and detailed event logs. However, process mining is not limited to information systems and can also be used to monitor other operational processes or systems. For example, we have applied process mining to complex X-ray machines, high-end copiers, web services, wafer steppers, careflows in hospitals, etc. All of these applications have in common that *there is a notion of a process* and that *the occurrence of activities are recorded in so-called event logs*.

Assuming that we are able to log events, a wide range of *process mining techniques* comes into reach. The basic idea of process mining is to learn from observed executions of a process and can be used to (1) *discover* new models (e.g., constructing a Petri net that is able to reproduce the observed behavior), (2) check the *conformance* of a model by checking whether the modeled behavior matches the observed behavior, and (3) *extend* an existing model by projecting information extracted from the logs onto some initial model (e.g., show bottlenecks in a process model by analyzing the event log). All three types of analysis

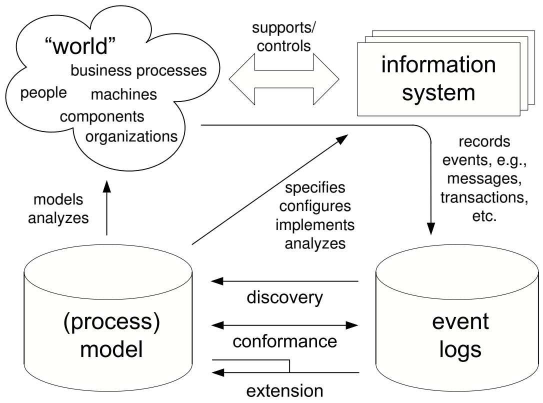


Fig. 1. Overview showing three types of process mining supported by ProM: (1) Discovery, (2) Conformance, and (3) Extension.

have in common that they assume the existence of some *event log*. Figure 1 shows the three types of process mining. Each of these is supported by ProM through various plug-ins as will be shown in the remainder using a running example.

3 Running Example

As a working example, we consider the "teleclaims" process of an Australian insurance company described in [2]. This process deals with the handling of inbound phone calls, whereby different types of insurance claims (household, car, etc.) are lodged over the phone. The process is supported by two separate call centres operating for two different organizational entities (Brisbane and Sydney). Both centres are similar in terms of incoming call volume (approx. 9,000 per week) and average total call handling time (550 seconds), but different in the way call centre agents are deployed, underlying IT systems, etc. The teleclaims process model is shown in Figure 2. The two highlighted boxes at the top show the subprocesses in both call centres. The lower part describes the process in the back-office.

This process model is expressed in terms of an Event-Driven Process Chain (EPC) (see [8] for a discussion on the semantics of EPCs). For the purpose of the paper it is not necessary to understand the process and EPC notation in any detail. However, for a basic understanding, consider the subprocess corresponding to the call centre in Brisbane. The process starts with *event* "Phone call received". This event triggers *function* "Check if sufficient information is available". This function is executed by a "Call Center Agent". Then a choice is made. The circle represents a so-called *connector*. The "x" inside the connector and the two outgoing arcs indicate that it is an exclusive OR-split (XOR). The XOR connector results in event "Sufficient information is available" or event "Sufficient information is not available". In the latter case the process ends. If the information is available, the claim is registered (cf. function "Register claim" also executed by a “Call Center Agent”) resulting in event “Claim is registered”. The call centre in Sydney has a similar subprocess and the back-office process should be self-explaining after this short introduction to EPCs. Note that there are three types of split and join connectors: AND, XOR, and OR, e.g., in the back-office process there is one AND-split (∧) indicating that the last part is executed in parallel.

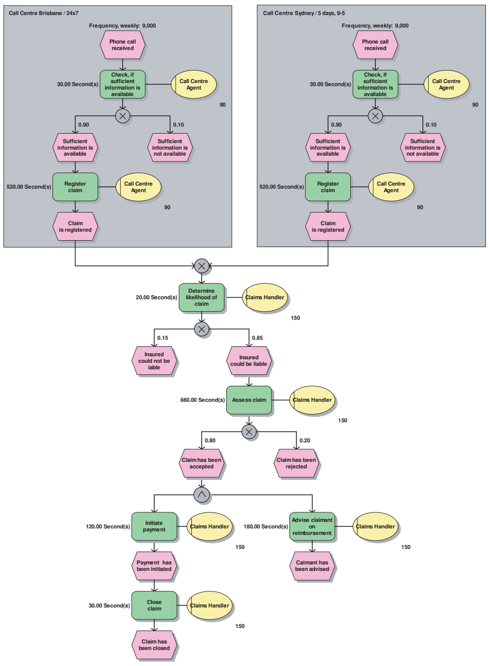


Fig. 2. Insurance claim handling EPC [2].

```
...
<ProcessInstance id="3055" description="Claim being handled">
  <AuditTrailEntry>
    <Data><Attribute name = "call centre">Sydney </Attribute>
    </Data><WorkflowModelElement>incoming claim
      </WorkflowModelElement>
    <EventType >complete</EventType>
    <Timestamp>2006-12-01T07:51:05.000+01:00</Timestamp>
    <Originator>customer</Originator>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <Data><Attribute name = "location">Sydney </Attribute>
    </Data><WorkflowModelElement>check if sufficient
      information is available</WorkflowModelElement>
    <EventType >start</EventType>
    <Timestamp>2006-12-01T07:51:05.000+01:00</Timestamp>
    <Originator>Call Centre Agent Sydney</Originator>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <Data><Attribute name = "location">Sydney </Attribute>
    </Data><WorkflowModelElement>check if sufficient
      information is available</WorkflowModelElement>
    <EventType >complete</EventType>
    <Timestamp>2006-12-01T07:51:25.000+01:00</Timestamp>
    <Originator>Call Centre Agent Sydney</Originator>
  </AuditTrailEntry>
  ...
  <AuditTrailEntry>
    <Data><Attribute name = "outcome">processed </Attribute>
    <Attribute name = "duration">1732 </Attribute>
    </Data><WorkflowModelElement>end</WorkflowModelElement>
    <EventType >complete</EventType>
    <Timestamp>2006-12-01T08:19:57.000+01:00</Timestamp>
    <Originator>Claims handler</Originator>
  </AuditTrailEntry>
</ProcessInstance>
...
```

Fig. 3. Fragment of the MXML log containing 3512 cases (process instances) and 46138 events (audit trail entries).

Figure 3 shows a fragment of the log in MXML format, the format used by ProM. In this case, the event log was obtained from a simulation using CPN Tools. Using ProMimport one can extract logs from a wide variety of systems, e.g., workflow management systems like Staffware, case handling systems like FLOWer, ERP components like PeopleSoft Financials, simulation tools like ARIS and CPN Tools, middleware systems like WebSphere, BI tools like ARIS PPM, etc., and it has also been used to develop many organization/system-specific conversions (e.g., hospitals, banks, governments, etc.). Figure 3 illustrates the typical data present in most event logs, i.e., a log is composed of process instances (i.e., cases) and within each instance there are audit trail entries (i.e., events) with various attributes. Note that it is not required that systems log all of this information, e.g., some systems do not record transactional information (e.g., just the completion of activities is recorded), related data, or timestamps. In the MXML format only the ProcessInstance (i.e., case) field and the WorkflowModelElement (i.e., activity) field are obligatory, i.e., *any event*

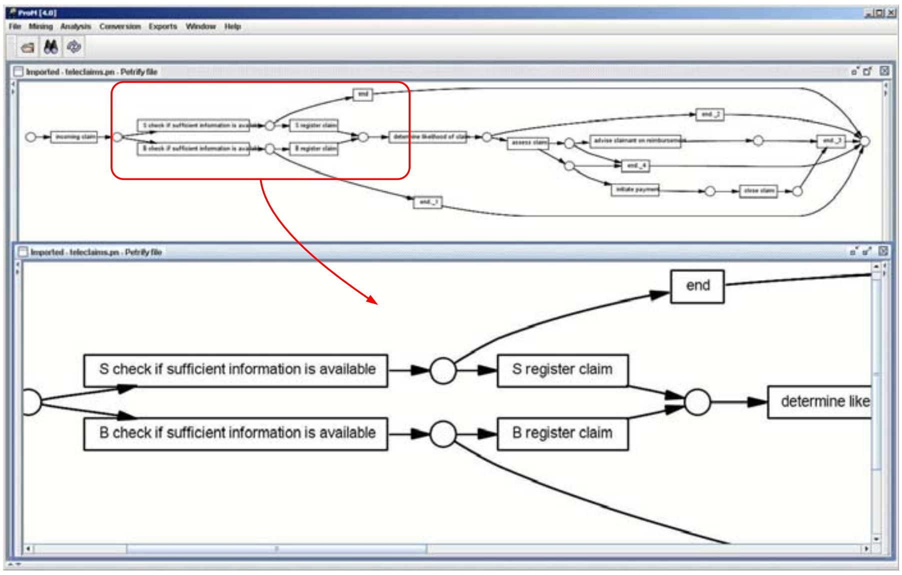

ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Imported - teleclaims.pn - Petrify file
incoming claim
S check if sufficient information is available
B check if sufficient information is available
S register claim
B register claim
end
end_1
end_2
end_3
end_4
determine likelihood of claim
assess claim
advise claimant on reimbursement
initiate payment
close claim
Imported - teleclaims.pn - Petrify file
end
S check if sufficient information is available
S register claim
determine like
B check if sufficient information is available
B register claim

Fig. 4. A Petri net discovered using ProM based on an analysis of the 3512 cases.

needs to be linked to a case (process instance) and an activity. All other fields (data, timestamps, resources, etc.) are optional.

For control-flow discovery, e.g., deriving a Petri net model from an MXML file, we often focus on the ordering of activities within individual cases. In this context, a single case σ can be described by a sequence of activities, i.e., a trace $\sigma \in A^*$ where A is the set of activities. Consequently, such an abstraction of the log can be described by a multiset of traces.

4 Discovery

Process mining techniques supporting *discovery* do not assume an a-priori model, i.e., based on an event log, some model is constructed (cf. Figure 1). ProM 4.0 offers 27 mining plug-ins able to construct a wide variety of models. One of the first plug-ins was the α-miner [5] which constructs a Petri net model from an MXML log, i.e., based on an analysis of the log which does not contain any explicit process information (e.g., AND/XOR-splits/joins), a process model is derived. However, the α-miner is unable to discover complex process models. For example, it is unable to *correctly* discover the teleclaims process illustrated in Figure 2. However, ProM 4.0 has several new mining plug-ins that are able to correctly discover this process using various approaches (regions, heuristics, genetic algorithms, etc.) and representations (Petri nets, EPCs, transitions systems, heuristic nets).

Figure 4 shows a Petri net discovered by ProM. The top window shows the overall process while the second window zooms in on the first part of the discovered model. This model is behaviorally equivalent to the EPC model in

Figure 2 and has been obtained using an approach which first builds a transition system (see Figure 5) and then uses extensions of the classical theory of regions [6] to construct a Petri net. ProM provides various ways to extract transition systems from logs, a plug-in to construct regions on-the-fly, and an import and export plug-in for Petrify [6] (see [3] for details).

Process mining is not limited to process models (i.e., control flow). ProM also allows for the discovery of models related to data, time, transactions, and resources. As an example, Figure 6 shows the plug-in to extract social networks from event logs using the technique presented in [1]. The social network shown in Figure 6 is constructed based on frequencies of work being transferred from one resource class to another. The diagram adequately shows that work is generated by customers and then flows via the call centre agents to the claims handlers in the back office.

It is impossible to provide an overview of all the discovery algorithms supported. However, of the 27 mining plug-ins we would like to mention the heuristics miner (Figure 7) able to discover processes in the presence of noise and the multi-phase miner using an EPC representation. Both approaches are more robust than the region-based approach and the classical α-algorithm. It is also possible to convert models of one type to another. For example, Figure 8 shows the EPC representation of the Petri net in Figure 4.

5 Conformance

Conformance checking requires, in addition to an event log, some a-priori model. This model may be handcrafted or obtained through process discovery. Whatever its source, ProM provides various ways of checking whether reality conforms to such a model. For example, there may be a process model indicating that purchase orders of more than one million Euro require two checks. Another example is the checking of the so-called “four-eyes principle”. Conformance checking may be used to detect deviations, to locate and explain these deviations, and to measure the severity of these deviations. ProM 4.0 also supports conformance checking, i.e., comparing an a-priori model with the observed reality stored in some MXML log. For example, we could take the discovered model shown in Figure 4 and compare it with the log shown in Figure 3 using the conformance checking plug-in in ProM. Figure 9 shows the result. This analysis shows that the fitness of the model is 1.0, i.e., the model is able to “parse” all cases. The conformance checker also calculates metrics such as behavioral appropriateness (i.e., precision) and structural appropriateness [9] all indicating that the discovered model is indeed a good reflection of reality. Note that, typically, conformance checking is done not with respect to a discovered model, but with respect to some normative/descriptive hand-crafted model. For example, given an event log obtained from the real teleclaims process it would be interesting to detect potential deviations from the process model in Figure 2. In case that there is not a complete a-priori process model but just a set of requirements (e.g., business rules), ProM’s LTL checker can be used.

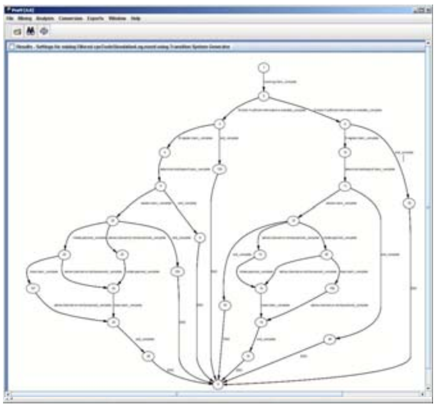
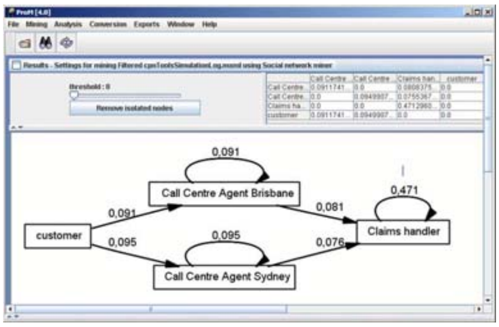


Fig. 5. Transition system system used to construct the Petri net in Figure 4.


ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Results - Settings for mining Filtered cpnToolsSimulationLog.mxml using Social network miner
threshold : 0
Remove isolated nodes
0,091
Call Centre Agent Brisbane
0,471
0,091
0,081
customer
0,095
0,095
0,076
Claims handler
Call Centre Agent Sydney

	Call Centre ...	Call Centre ...	Claims han...	customer
Call Centre ...	0.0911741 ...	0.0	0.0808375 ...	0.0
Call Centre ...	0.0	0.0949907 ...	0.0755367 ...	0.0
Claims ha...	0.0	0.0	0.4712960 ...	0.0
customer	0.0911741 ...	0.0949907 ...	0.0	0.0

Fig. 6. Social network obtained using the "handover of work" metric.

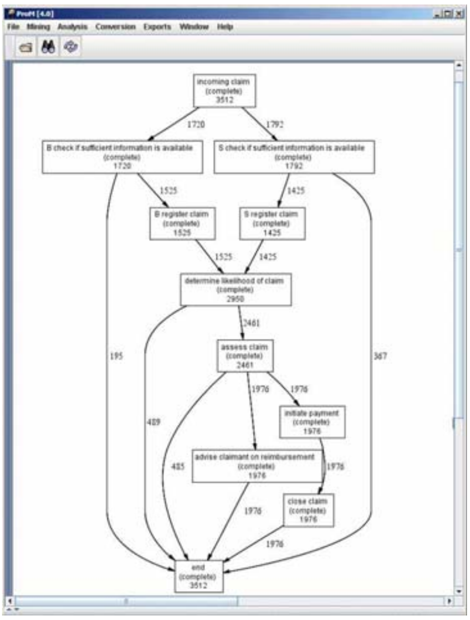


Fig. 7. Heuristics net obtained by applying the heuristics miner to the log of Figure 3.

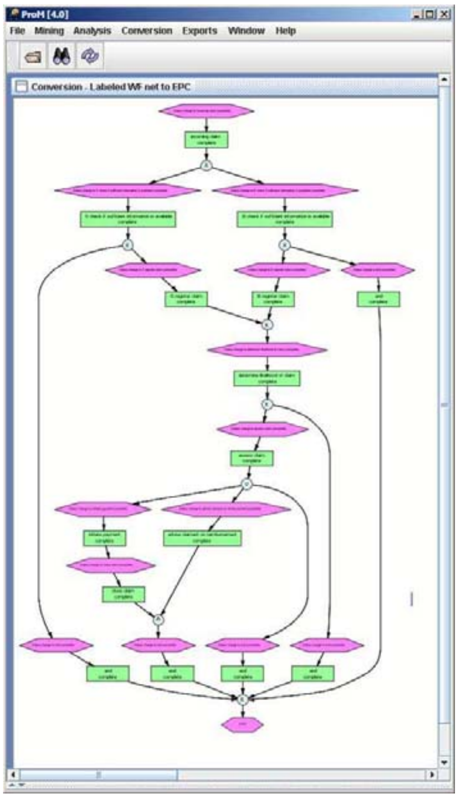

ProM [4.0]
File Mining Analysis Conversion Exports Window Help
Conversion - Labeled WF net to EPC

Fig. 8. EPC discovered from the log in Figure 3.

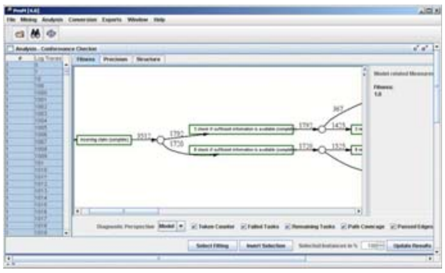


Fig. 9. Conformance checker.

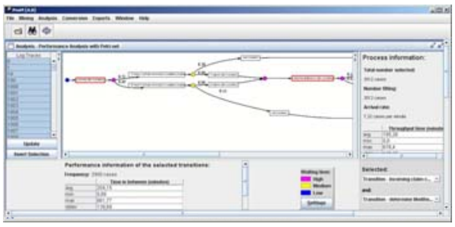

File Mining Analysis Conversion Exports Window Help
Analysis - Performance Analysis with Petri net
Log Traces
Update
Invert Selection
Process information:
Total number selected:
Number fitting:
Arrival rate:
Throughput time (minutes)
avg
min
max
Performance information of the selected transitions:
Frequency:
Time in between (minutes)
avg
min
max
stdev
Waiting time:
High
Medium
Low
Settings
Selected:
and:

Fig. 10. Performance analyzer.

6 Extension

For model *extension* it is also assumed that there is an initial model (cf. Figure 1). This model is extended with a new aspect or perspective, i.e., the goal is not to check conformance but to enrich the model with performance/time aspects, organizational/resource aspects, and data/information aspects. Consider for example a Petri net (either discovered, hand-crafted, or resulting from some model transformation) describing a process which is also logged. It is possible to enrich the Petri net using information in the log. Most logs also contain information about resources, data, and time. ProM 4.0 supports for example decision mining, i.e., by analyzing the data attached to events and using classical decision tree analysis, it is possible to add decision rules to the Petri net (represented as conditions on arcs). Information about resources (Originator field in the MXML log) can be analyzed and used to add allocation rules to a Petri net. Figure 10 shows a performance analysis plug-in which projects timing information on places and transitions. It graphically shows the bottlenecks and all kinds of performance indicators, e.g., average/variance of the total flow time or the time spent between two activities. The information coming from all kinds of sources can be stitched together and exported to CPN Tools, i.e., ProM is able to turn MXML logs into colored Petri nets describing all perspectives (control-flow, data, time, resources, etc.). CPN Tools can then be used to simulate the process without adding any additional information to the generated model.

7 Additional Functionality

It is not possible to give a complete overview of all 142 plug-ins. The figures shown in previous sections reflect only the functionality of 7 plug-ins. However, it is important to note that the functionality of ProM is not limited to process mining. ProM also allows for *model conversion.* For example, a model discovered in terms of a heuristic net can be mapped onto an EPC which can be converted into a Petri net which is saved as a YAWL file that can be uploaded in the workflow system YAWL thereby directly enacting the discovered model. For some of the models, ProM also provides *analysis* plug-ins. For example, the basic Petri net analysis techniques (invariants, reachability graphs, reduction rules, S-components, soundness checks, etc.) are supported. There are also interfaces to different analysis (e.g., Petrify, Fiona, and Woflan) and visualization (e.g., FSMView and DiaGraphica) tools.

8 Conclusion

ProM 4.0 consolidates the state-of-the-art of process mining. It provides a plug-able environment for process mining offering a wide variety of plug-ins for process discovery, conformance checking, model extension, model transformation, etc. ProM is open source and can be downloaded from www.processmining.org. Many of its plug-ins work on Petri nets, e.g., there are several plug-ins to discover Petri nets using techniques ranging from genetic algorithms and heuristics to regions and partial orders. Moreover, Petri nets can be analyzed in various ways using the various analysis plug-ins.

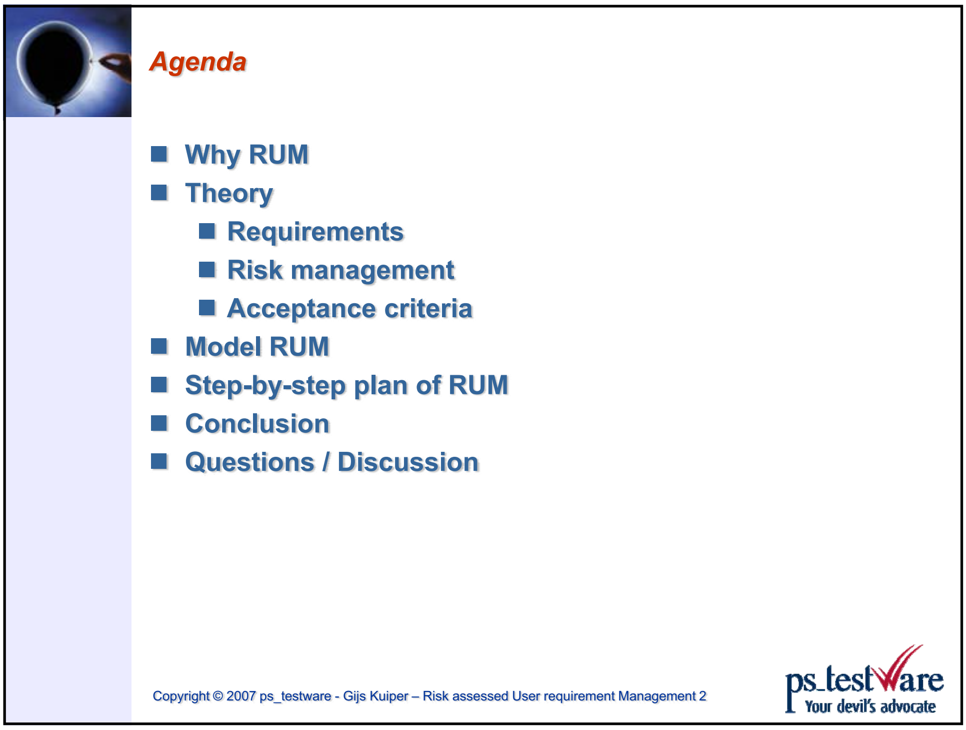
Acknowledgements
The development of ProM is supported by EIT, NWO-EW, the Technology Foundation STW, and the IOP program of the Dutch Ministry of Economic Affairs.

References

1. W.M.P. van der Aalst, H.A. Reijers, and M. Song. Discovering Social Networks from Event Logs. *Computer Supported Cooperative work*, 14(6):549–593, 2005.
2. W.M.P. van der Aalst, M. Rosemann, and M. Dumas. Deadline-based Escalation in Process-Aware Information Systems. *Decision Support Systems*, 2007 (to appear).
3. W.M.P. van der Aalst, V. Rubin, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach using Transition Systems and Regions. BPM Center Report BPM-06-30, BPMcenter.org, 2006.
4. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
5. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
6. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.
7. B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
8. E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. *Data and Knowledge Engineering*, 56(1):23–40, 2006.
9. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2006.

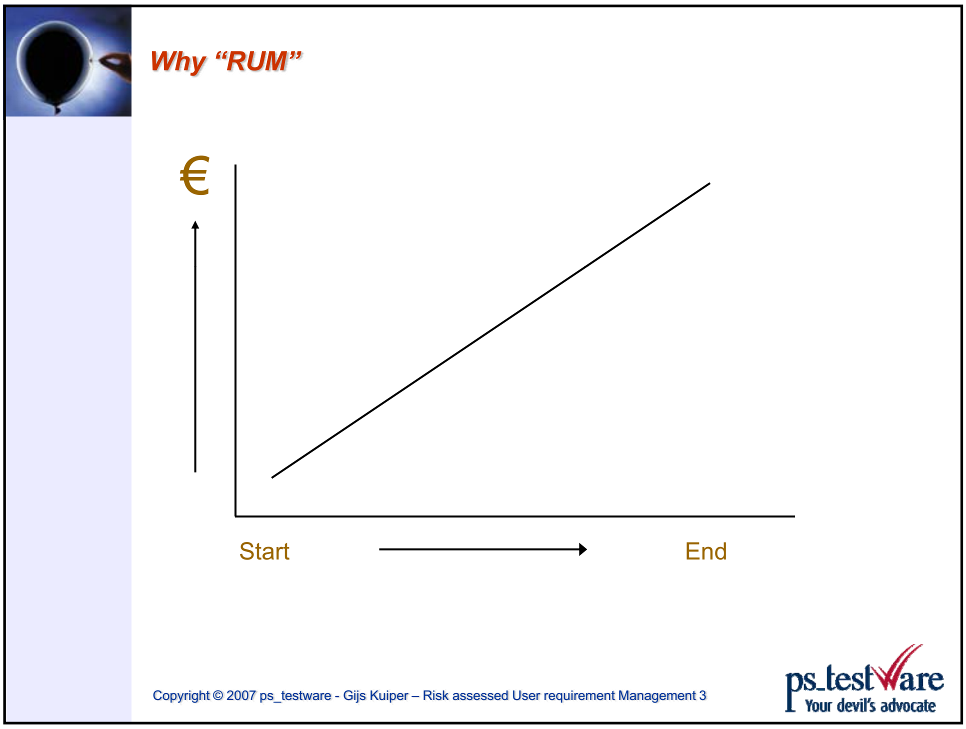
RUM^{TM}

Risk assessed User requirements Management

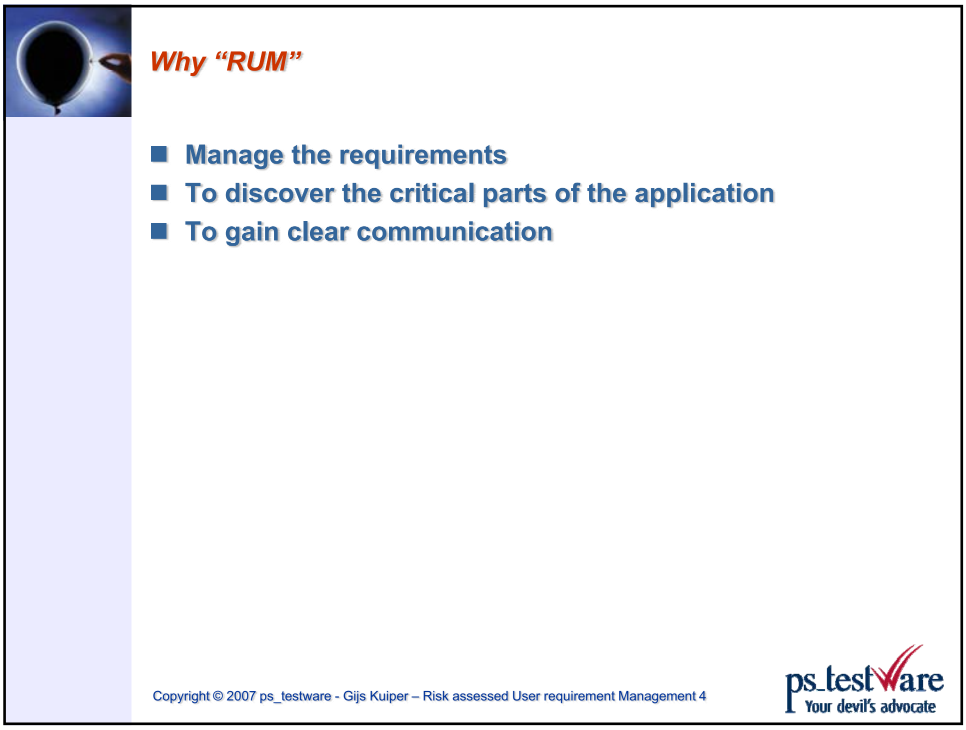


Agenda

- Why RUM
- Theory
  - Requirements
  - Risk management
  - Acceptance criteria
- Model RUM
- Step-by-step plan of RUM
- Conclusion
- Questions / Discussion

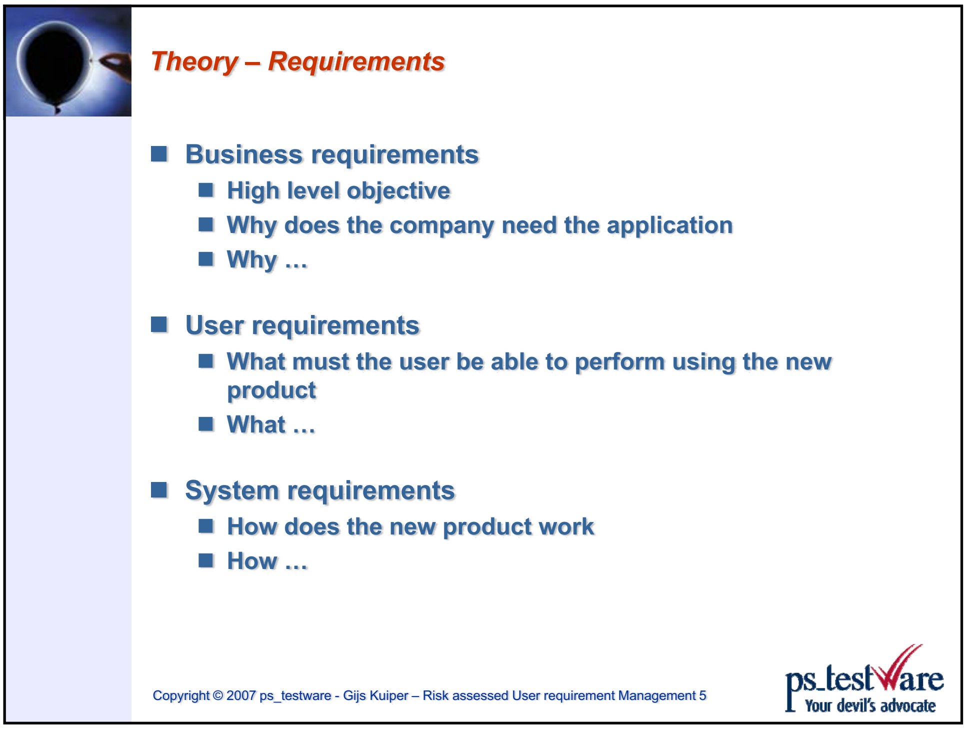


Why "RUM"



Why "RUM"

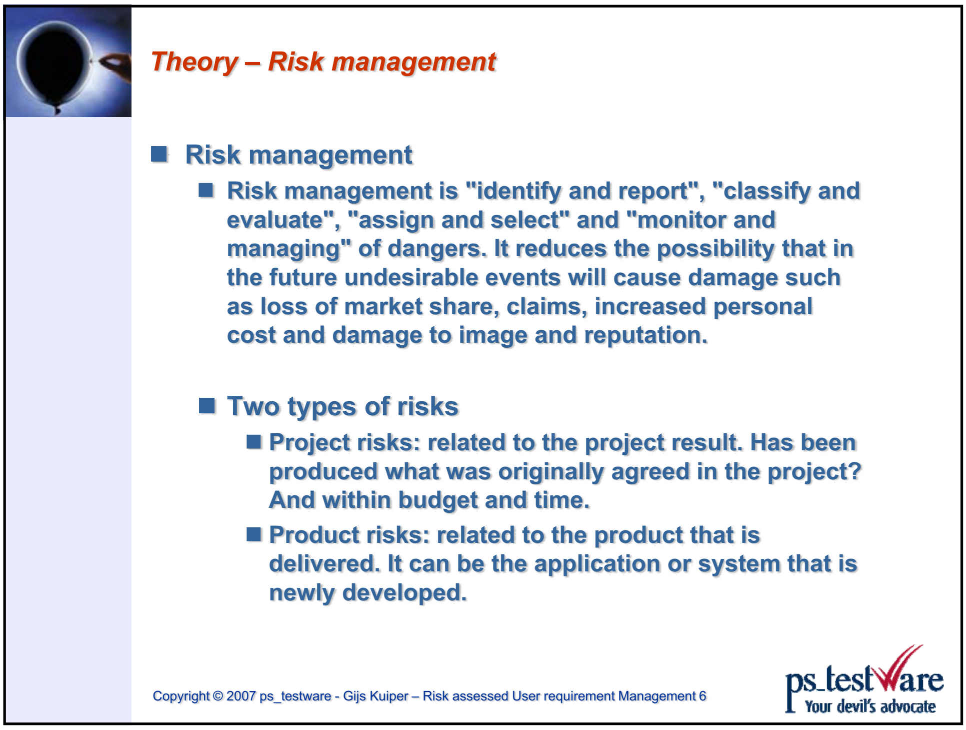
- Manage the requirements
- To discover the critical parts of the application
- To gain clear communication



Theory – Requirements

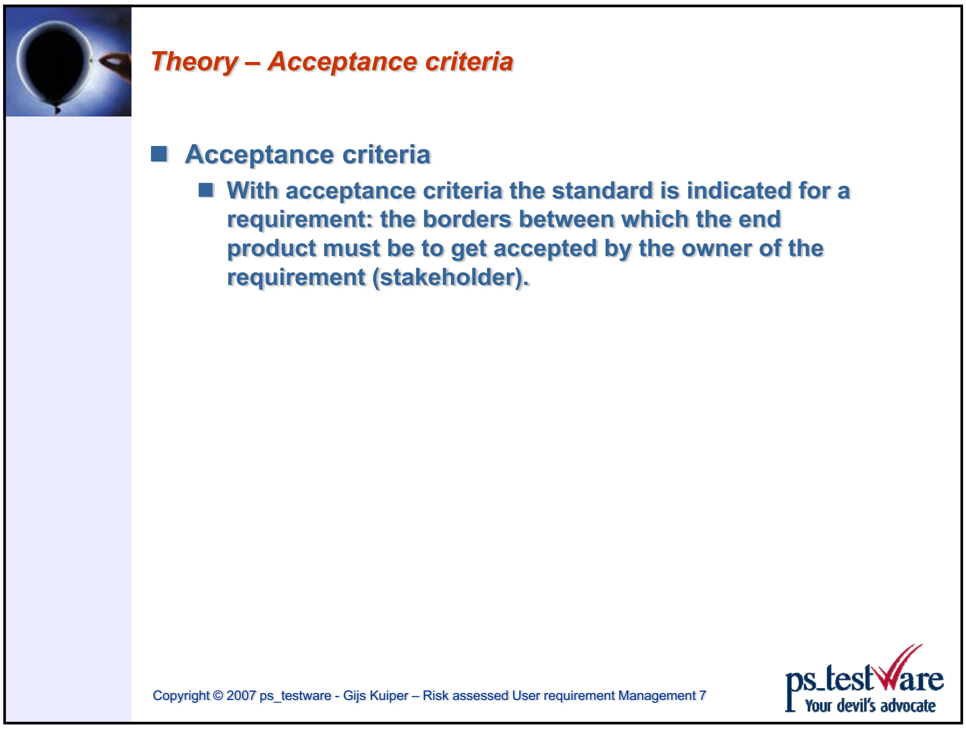
- Business requirements
  - High level objective
  - Why does the company need the application
  - Why …
- User requirements
  - What must the user be able to perform using the new product
  - What …
- System requirements
  - How does the new product work
  - How …

ps_testware
Your devil's advocate



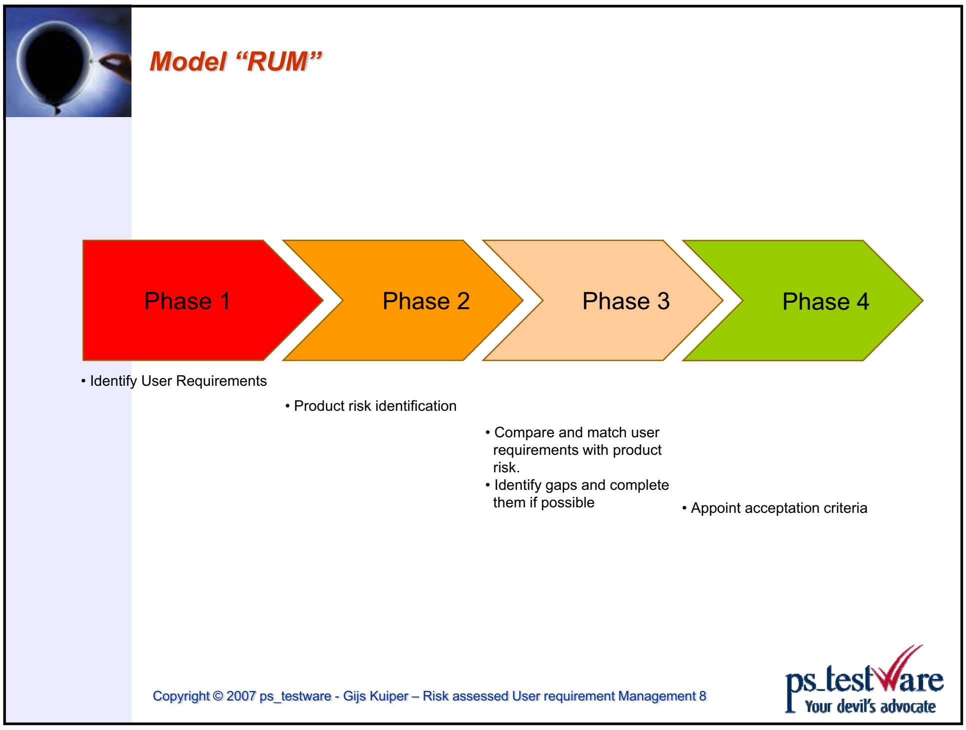
Theory – Risk management

- Risk management
  - Risk management is "identify and report", "classify and evaluate", "assign and select" and "monitor and managing" of dangers. It reduces the possibility that in the future undesirable events will cause damage such as loss of market share, claims, increased personal cost and damage to image and reputation.
  - Two types of risks
    - Project risks: related to the project result. Has been produced what was originally agreed in the project? And within budget and time.
    - Product risks: related to the product that is delivered. It can be the application or system that is newly developed.



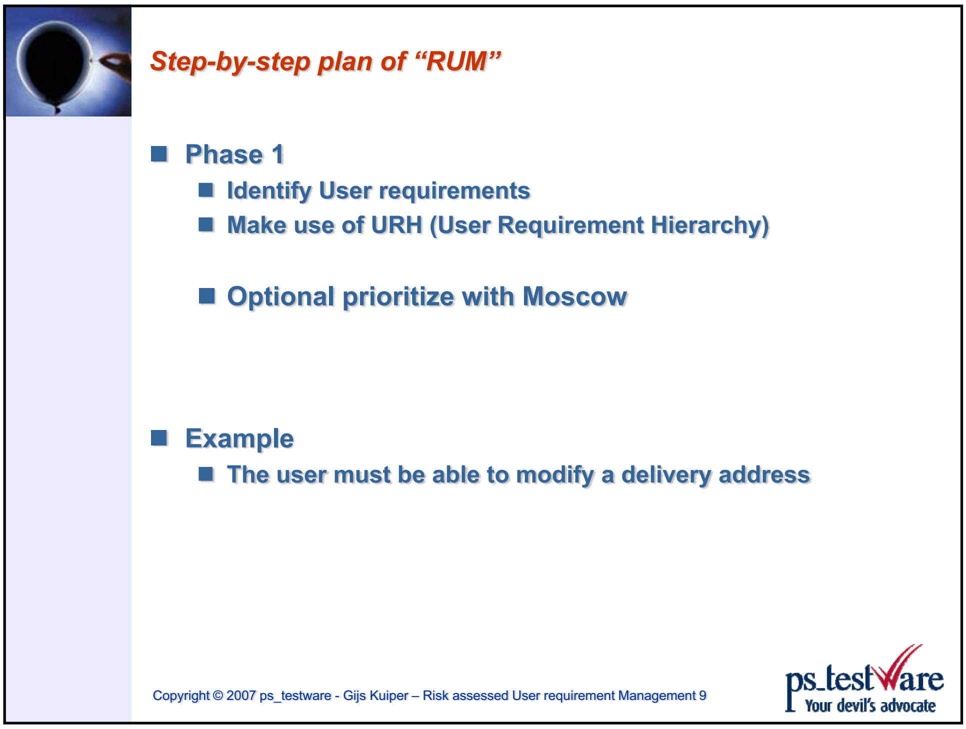
Theory – Acceptance criteria

- Acceptance criteria
  - With acceptance criteria the standard is indicated for a requirement: the borders between which the end product must be to get accepted by the owner of the requirement (stakeholder).



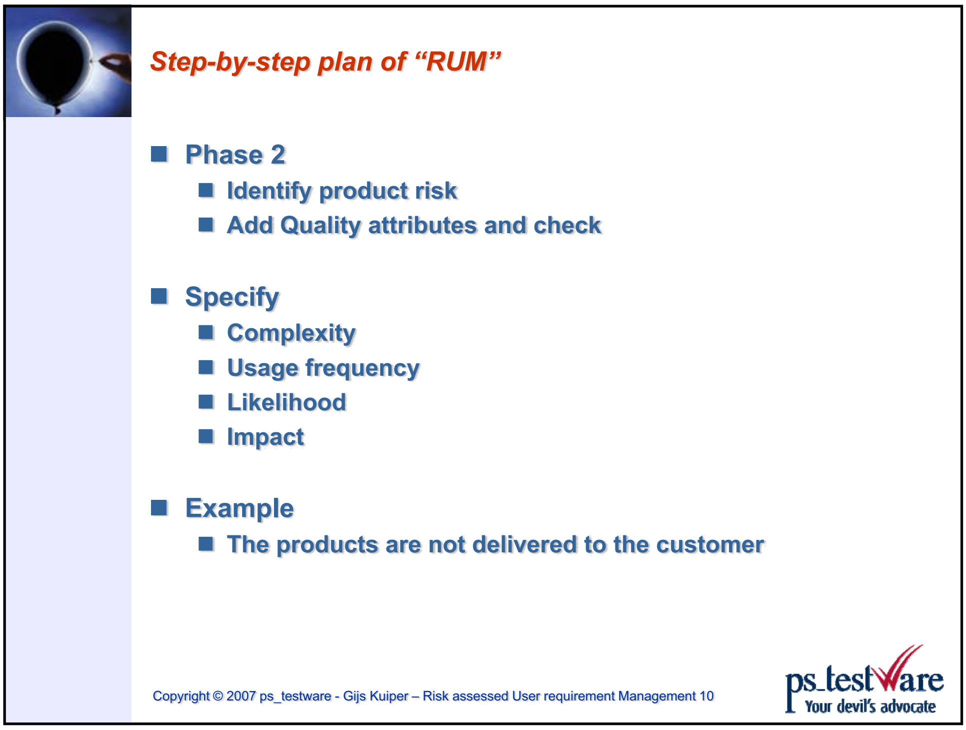
Model "RUM"

- Identify User Requirements
- Product risk identification
- Compare and match user requirements with product risk.
- Identify gaps and complete them if possible
- Appoint acceptation criteria



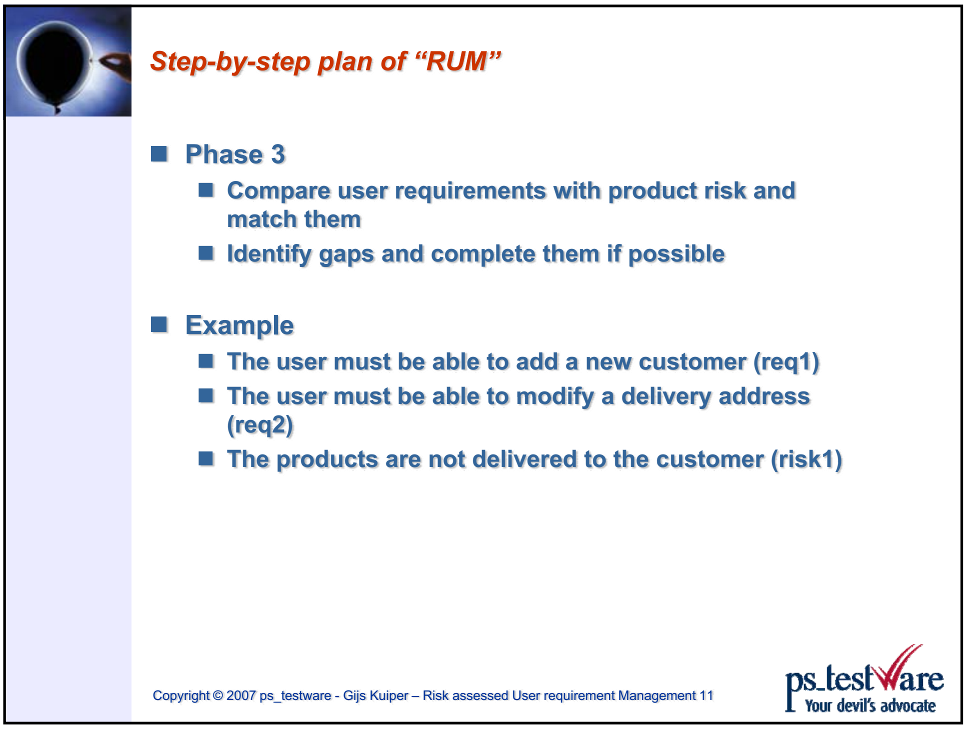
Step-by-step plan of "RUM"

- Phase 1
  - Identify User requirements
  - Make use of URH (User Requirement Hierarchy)
  - Optional prioritize with Moscow
- Example
  - The user must be able to modify a delivery address



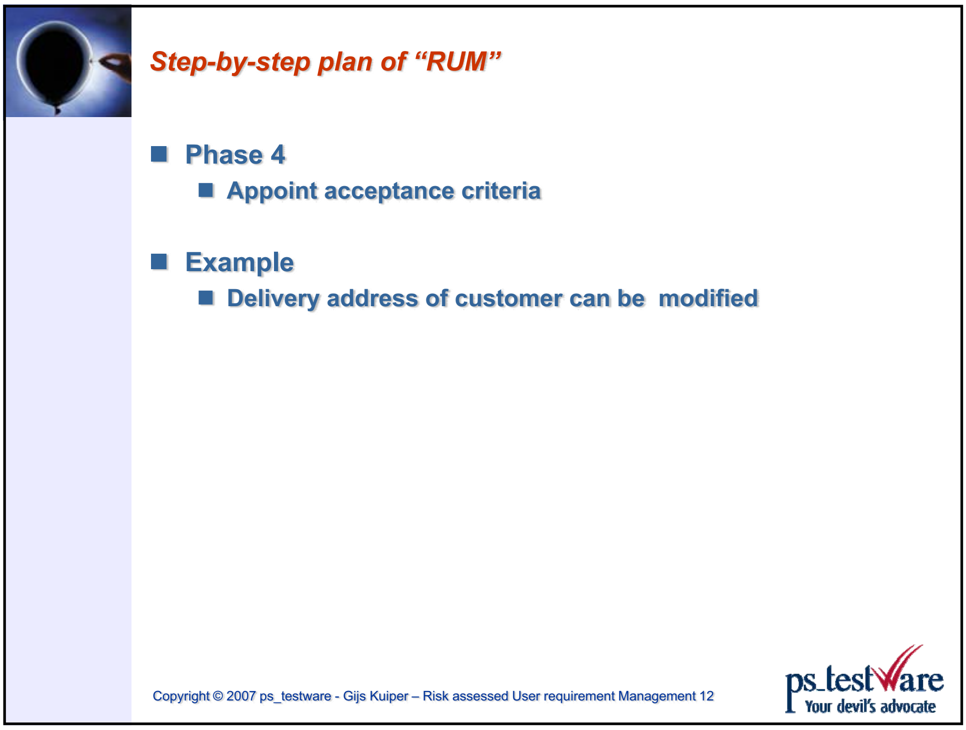
Step-by-step plan of "RUM"

- Phase 2
  - Identify product risk
  - Add Quality attributes and check
- Specify
  - Complexity
  - Usage frequency
  - Likelihood
  - Impact
- Example
  - The products are not delivered to the customer



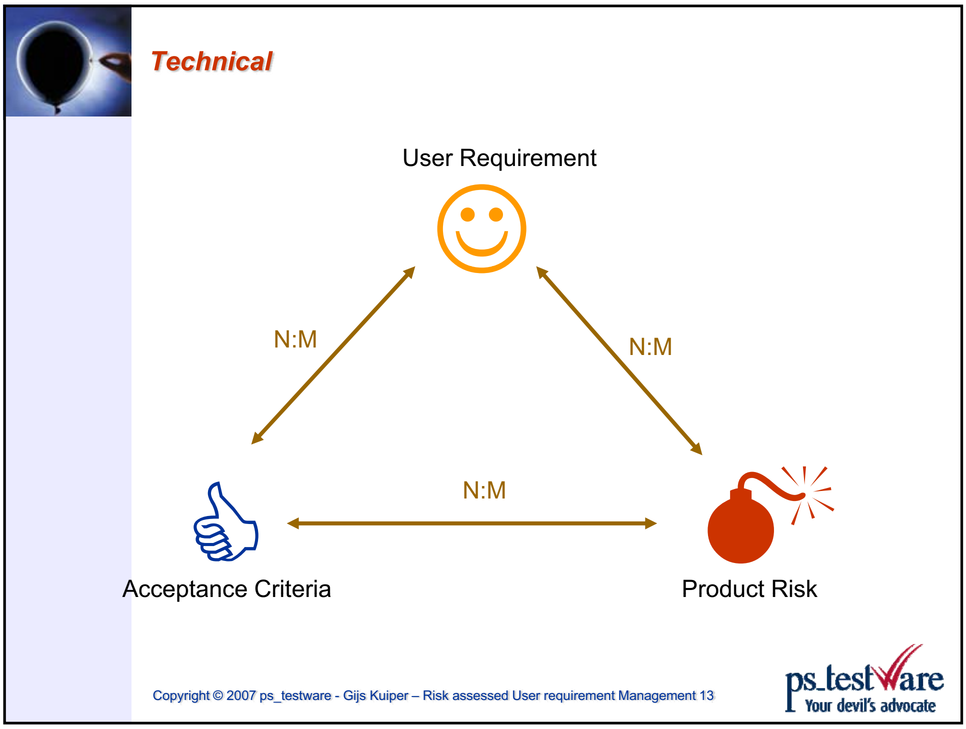
Step-by-step plan of “RUM”

- Phase 3
  - Compare user requirements with product risk and match them
  - Identify gaps and complete them if possible
- Example
  - The user must be able to add a new customer (req1)
  - The user must be able to modify a delivery address (req2)
  - The products are not delivered to the customer (risk1)



Step-by-step plan of “RUM”

- Phase 4
  - Appoint acceptance criteria
- Example
  - Delivery address of customer can be modified



Technical

User Requirement

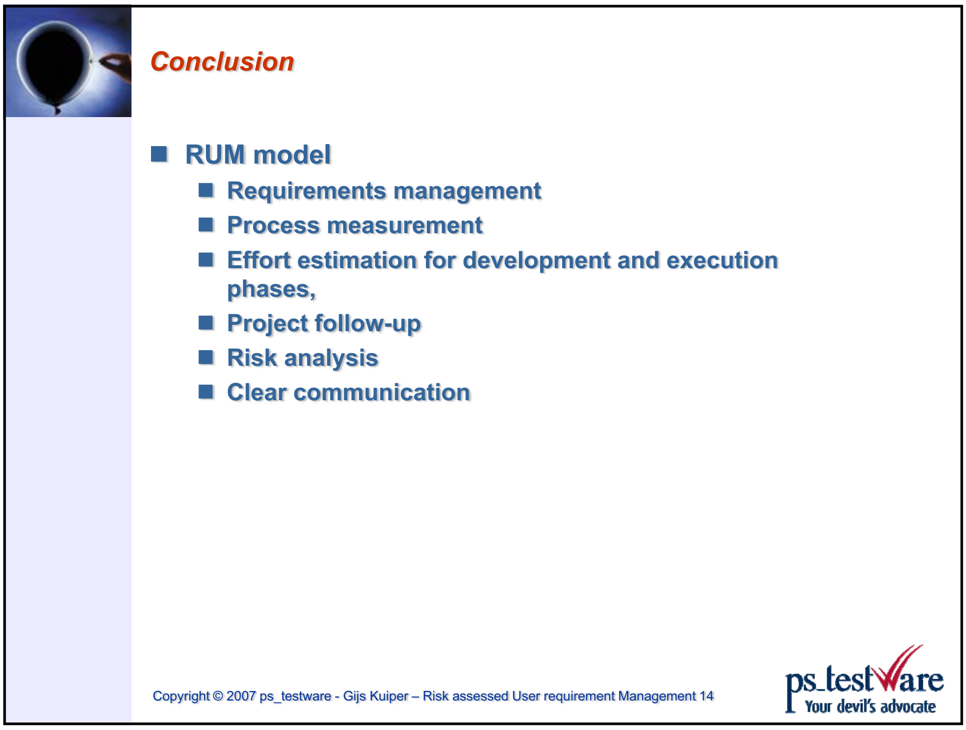
N:M

N:M

N:M

Acceptance Criteria

Product Risk



Conclusion

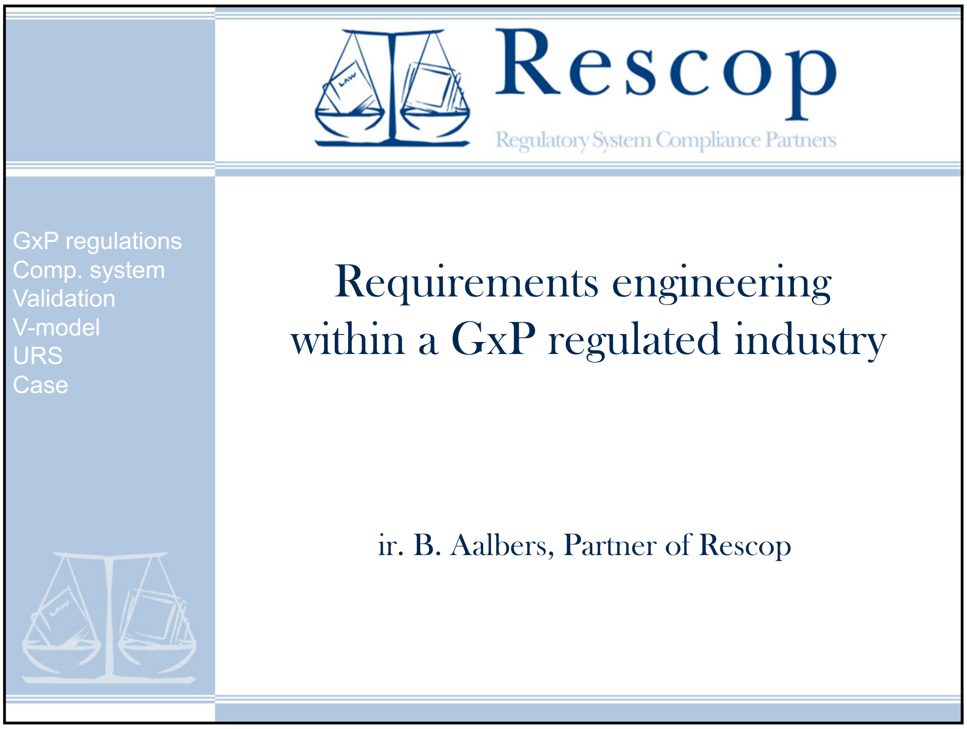
- RUM model
  - Requirements management
  - Process measurement
  - Effort estimation for development and execution phases,
  - Project follow-up
  - Risk analysis
  - Clear communication

Questions / Discussion

?

Gijs Kuiper
gijs.kuiper@pstestware.com

ps_testware
G. Stephensonweg 14
4207 HB Gorinchem
The Netherlands
www.pstestware.com



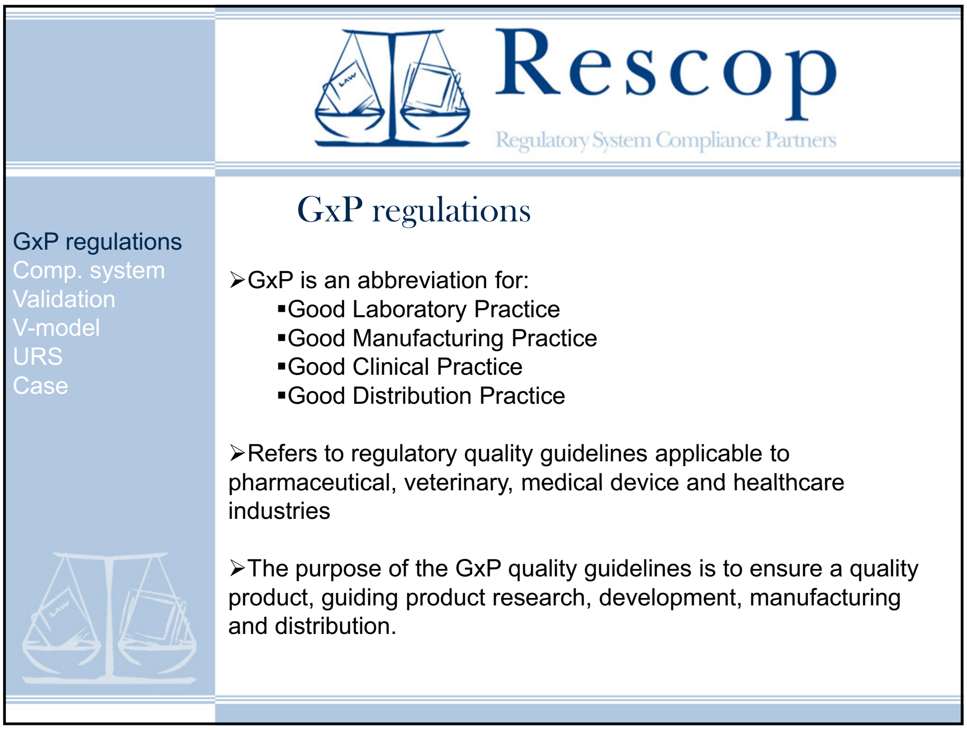
Rescop

Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

Requirements engineering within a GxP regulated industry

ir. B. Aalbers, Partner of Rescop



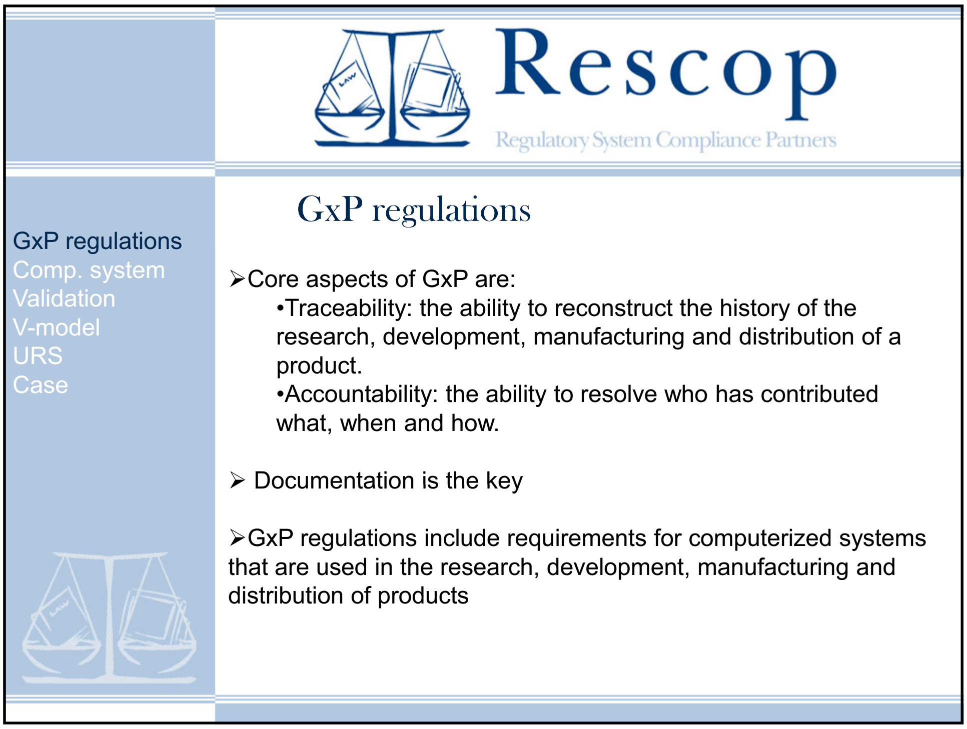
Rescop

Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

GxP regulations

- GxP is an abbreviation for:
  - Good Laboratory Practice
  - Good Manufacturing Practice
  - Good Clinical Practice
  - Good Distribution Practice
- Refers to regulatory quality guidelines applicable to pharmaceutical, veterinary, medical device and healthcare industries
- The purpose of the GxP quality guidelines is to ensure a quality product, guiding product research, development, manufacturing and distribution.



Rescop
Regulatory System Compliance Partners

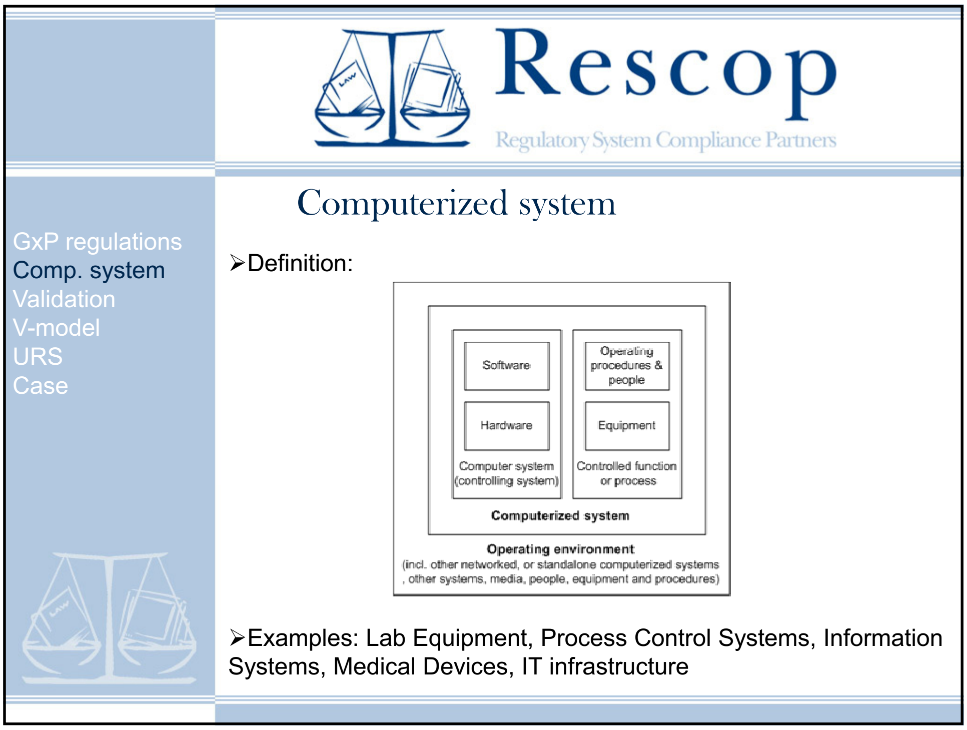
GxP regulations
Comp. system
Validation
V-model
URS
Case

GxP regulations

➢Core aspects of GxP are:
- •Traceability: the ability to reconstruct the history of the research, development, manufacturing and distribution of a product.
- •Accountability: the ability to resolve who has contributed what, when and how.

➢ Documentation is the key

➢GxP regulations include requirements for computerized systems that are used in the research, development, manufacturing and distribution of products



Rescop
Regulatory System Compliance Partners

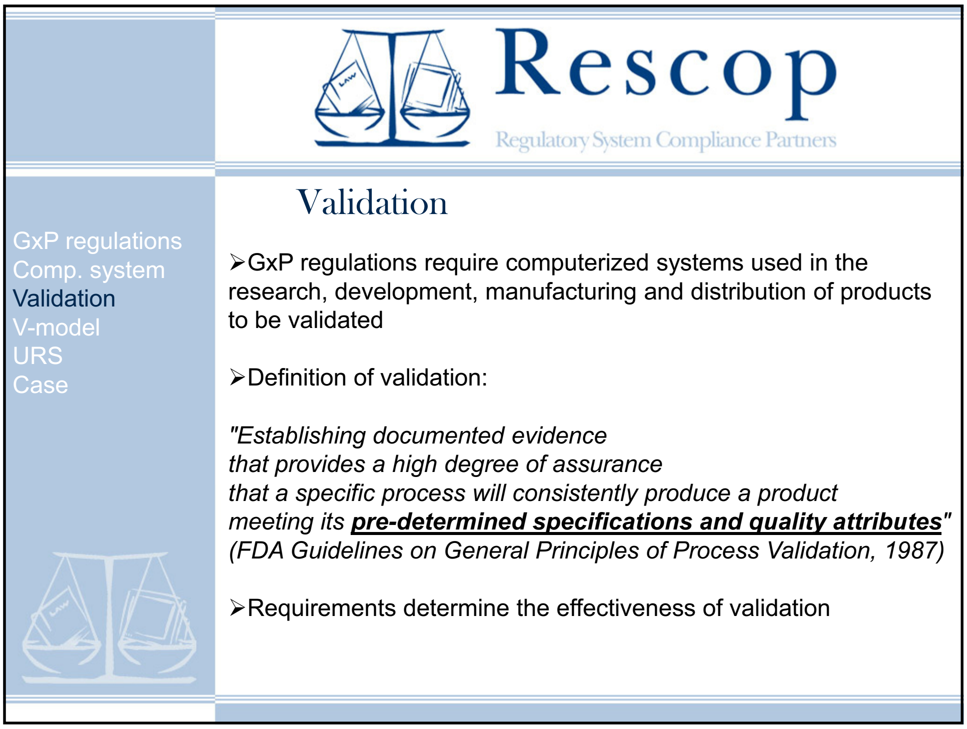
GxP regulations
Comp. system
Validation
V-model
URS
Case

Computerized system

➢Definition:

Software
Hardware
Computer system (controlling system)
Operating procedures & people
Equipment
Controlled function or process
Computerized system
Operating environment
(incl. other networked, or standalone computerized systems , other systems, media, people, equipment and procedures)

➢Examples: Lab Equipment, Process Control Systems, Information Systems, Medical Devices, IT infrastructure



Rescop

Regulatory System Compliance Partners

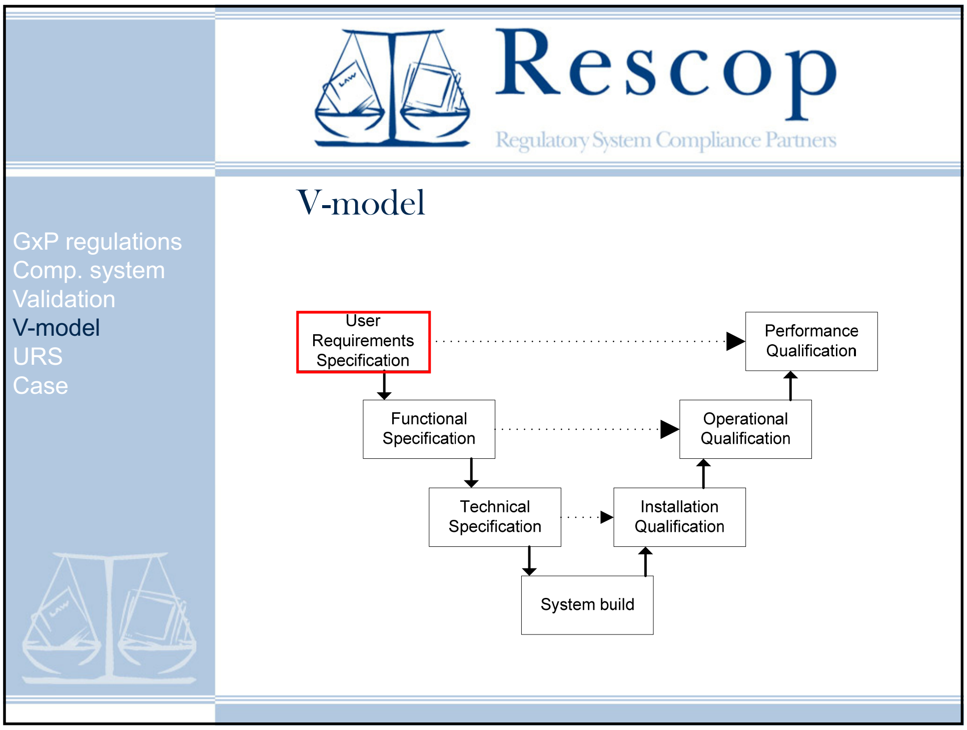
GxP regulations
Comp. system
Validation
V-model
URS
Case

Validation

- GxP regulations require computerized systems used in the research, development, manufacturing and distribution of products to be validated
- Definition of validation:

"Establishing documented evidence
that provides a high degree of assurance
that a specific process will consistently produce a product
meeting its ***pre-determined specifications and quality attributes****"*
(FDA Guidelines on General Principles of Process Validation, 1987)

- Requirements determine the effectiveness of validation



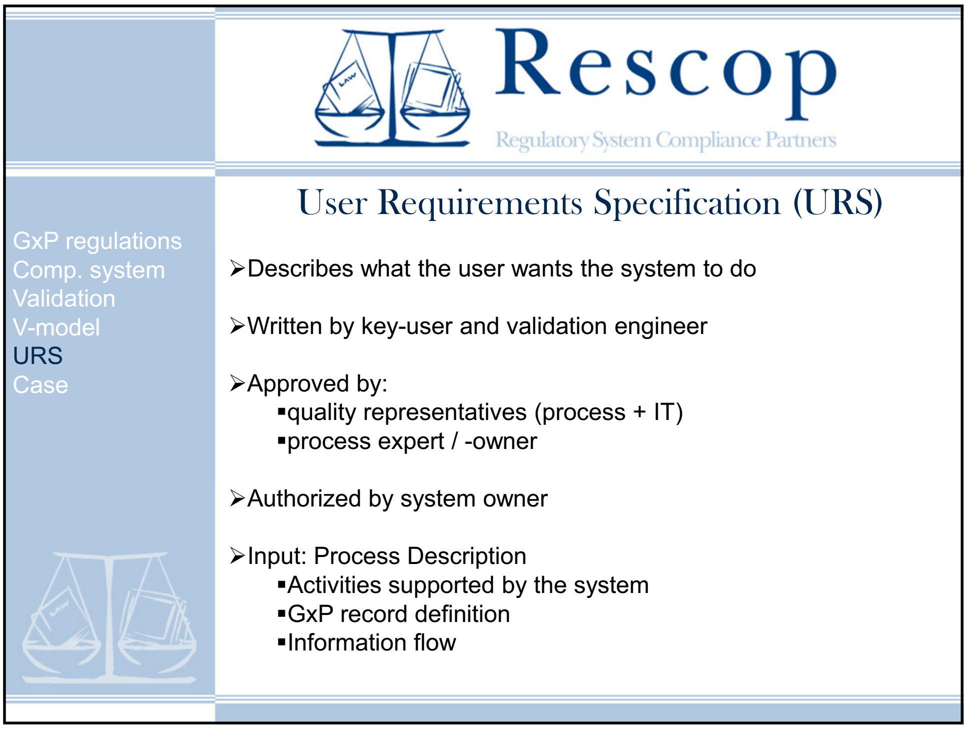
Rescop

Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

V-model

User Requirements Specification
Performance Qualification
Functional Specification
Operational Qualification
Technical Specification
Installation Qualification
System build



Rescop

Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

User Requirements Specification (URS)

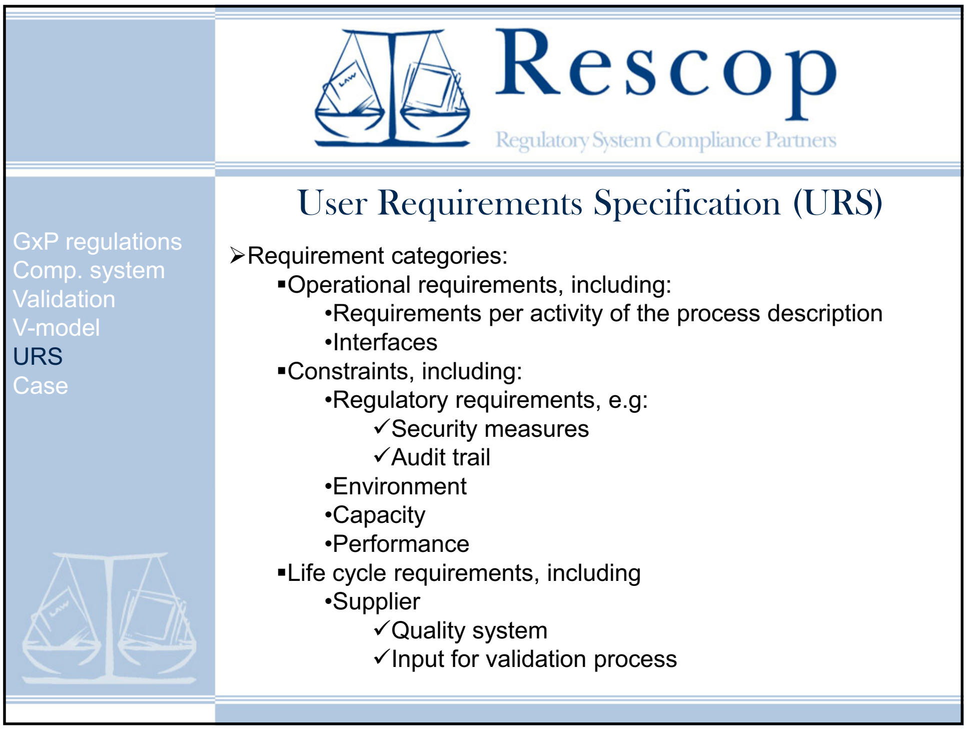
➢Describes what the user wants the system to do

➢Written by key-user and validation engineer

➢Approved by:
- quality representatives (process + IT)
- process expert / -owner

➢Authorized by system owner

➢Input: Process Description
- Activities supported by the system
- GxP record definition
- Information flow



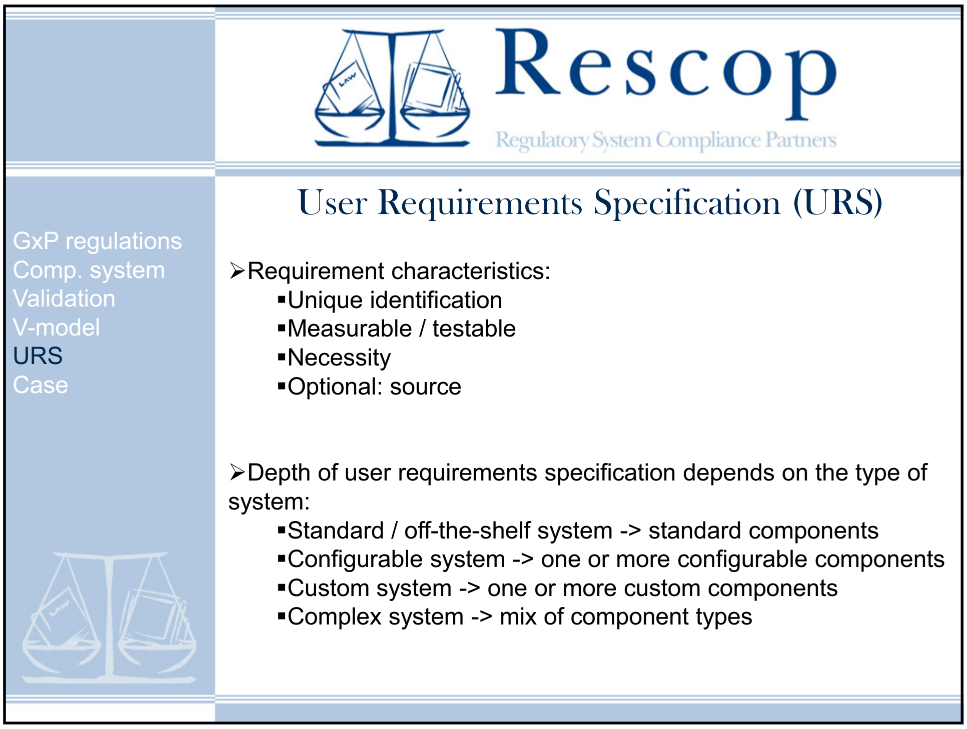
Rescop

Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

User Requirements Specification (URS)

➢Requirement categories:
- Operational requirements, including:
  - Requirements per activity of the process description
  - Interfaces
- Constraints, including:
  - Regulatory requirements, e.g:
    - ✓Security measures
    - ✓Audit trail
  - Environment
  - Capacity
  - Performance
- Life cycle requirements, including
  - Supplier
    - ✓Quality system
    - ✓Input for validation process



Rescop

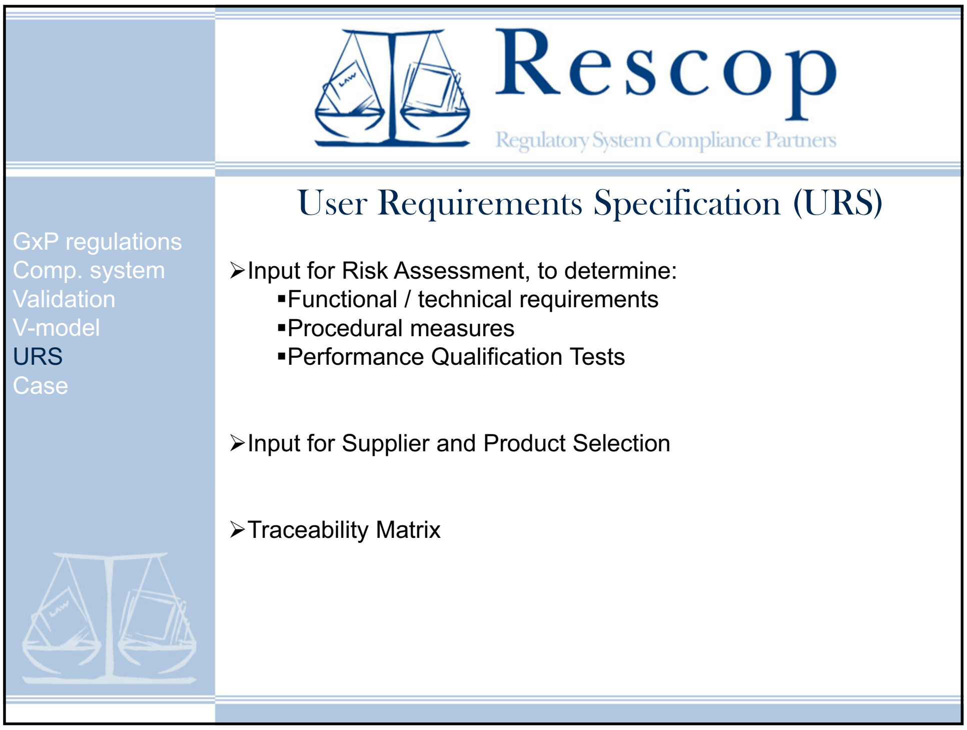
Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

User Requirements Specification (URS)

- Requirement characteristics:
  - Unique identification
  - Measurable / testable
  - Necessity
  - Optional: source

- Depth of user requirements specification depends on the type of system:
  - Standard / off-the-shelf system -> standard components
  - Configurable system -> one or more configurable components
  - Custom system -> one or more custom components
  - Complex system -> mix of component types



Rescop

Regulatory System Compliance Partners

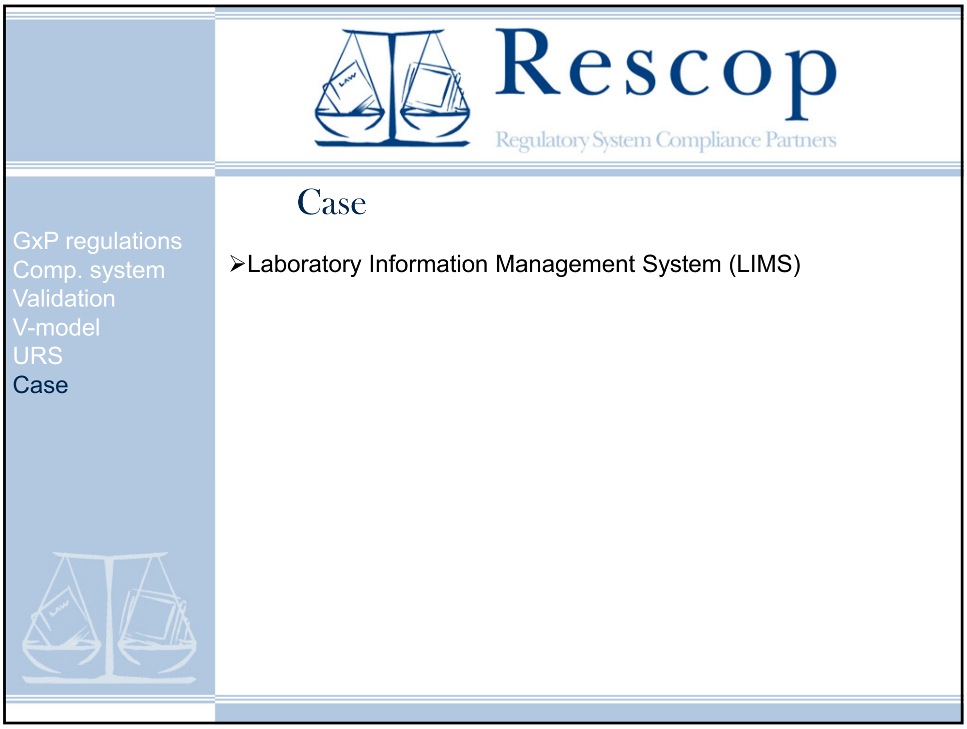
GxP regulations
Comp. system
Validation
V-model
URS
Case

User Requirements Specification (URS)

- Input for Risk Assessment, to determine:
  - Functional / technical requirements
  - Procedural measures
  - Performance Qualification Tests

- Input for Supplier and Product Selection

- Traceability Matrix



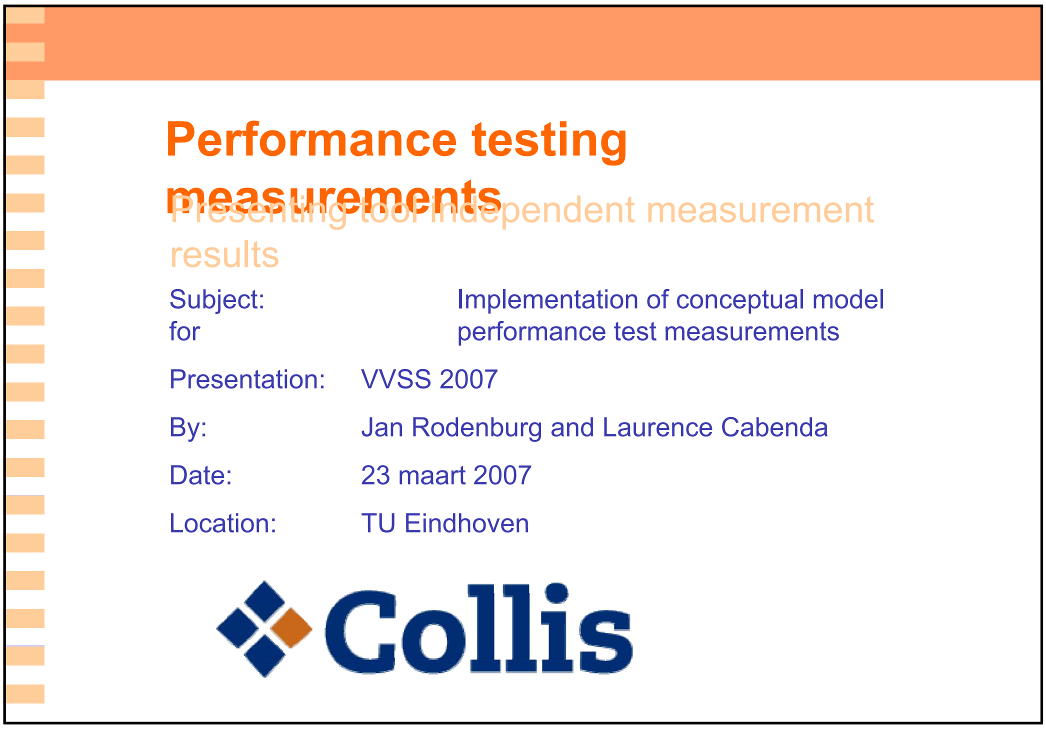
Rescop

Regulatory System Compliance Partners

GxP regulations
Comp. system
Validation
V-model
URS
Case

Case

- Laboratory Information Management System (LIMS)



Performance testing measurements

Presenting tool independent measurement results

Subject: Implementation of conceptual model
for performance test measurements

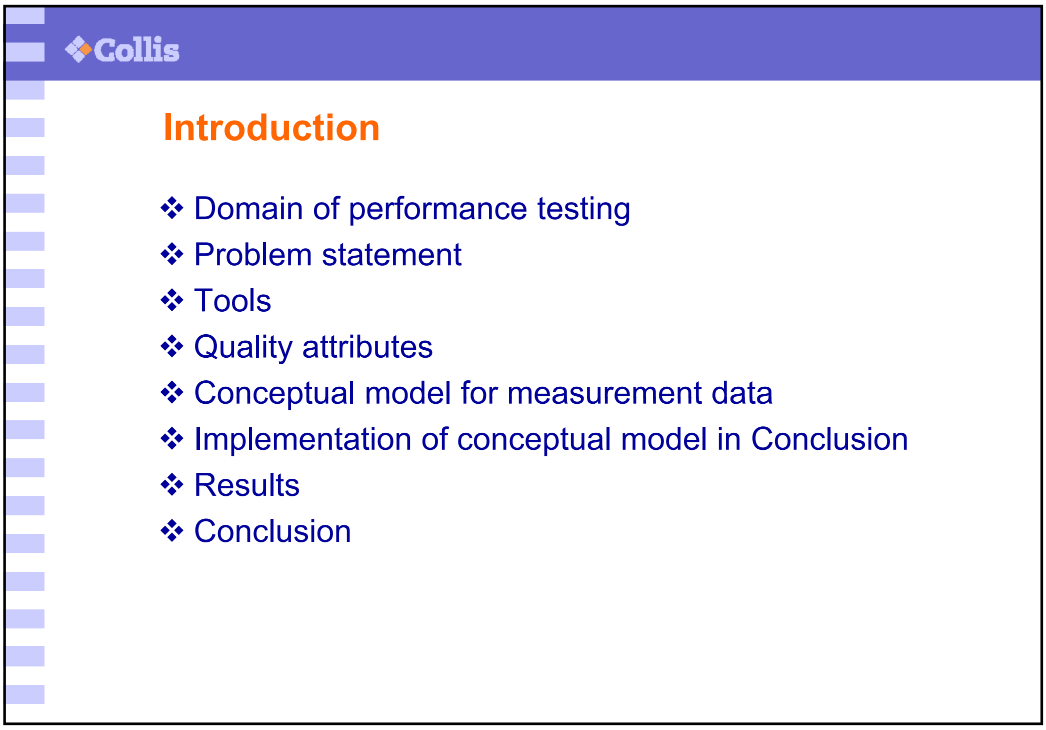
Presentation: VVSS 2007

By: Jan Rodenburg and Laurence Cabenda

Date: 23 maart 2007

Location: TU Eindhoven

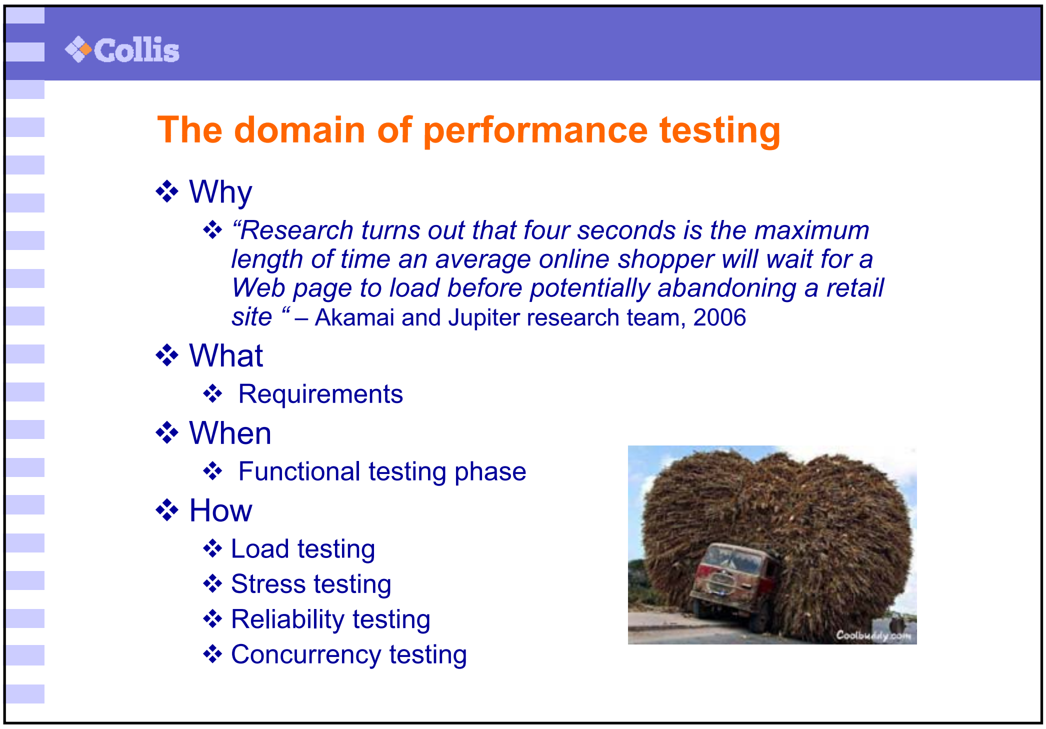
Collis



Collis

Introduction

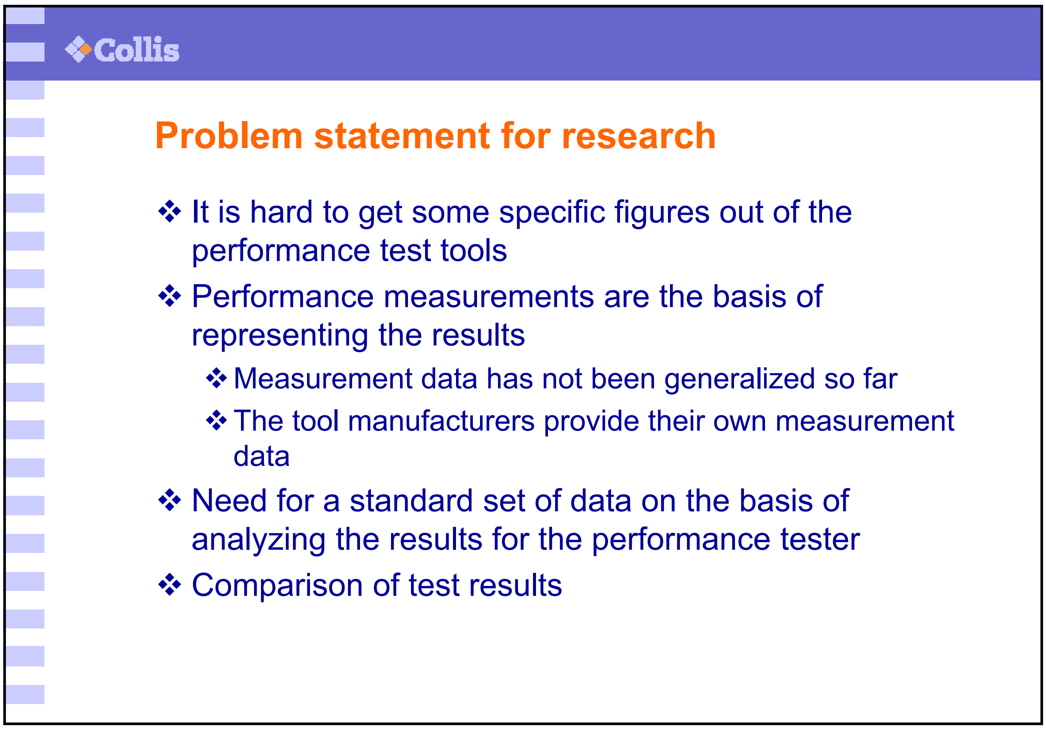
- ❖ Domain of performance testing
- ❖ Problem statement
- ❖ Tools
- ❖ Quality attributes
- ❖ Conceptual model for measurement data
- ❖ Implementation of conceptual model in Conclusion
- ❖ Results
- ❖ Conclusion



The domain of performance testing

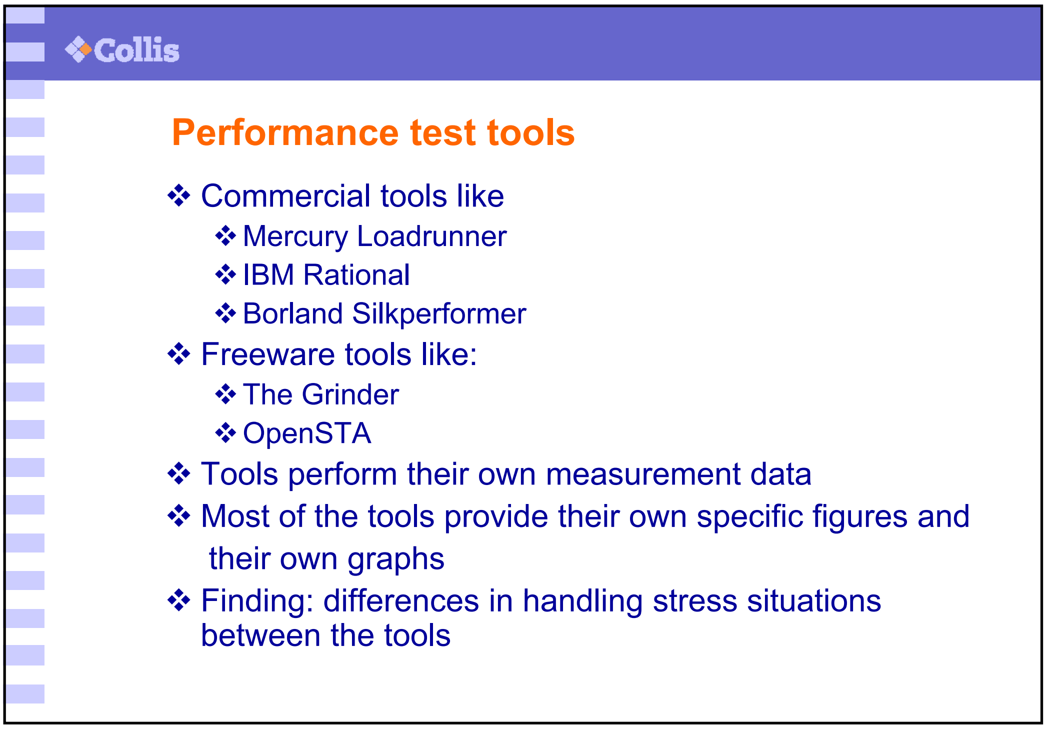
- Why
  - *"Research turns out that four seconds is the maximum length of time an average online shopper will wait for a Web page to load before potentially abandoning a retail site "* – Akamai and Jupiter research team, 2006
- What
  - Requirements
- When
  - Functional testing phase
- How
  - Load testing
  - Stress testing
  - Reliability testing
  - Concurrency testing

Coolbuddy.com



Problem statement for research

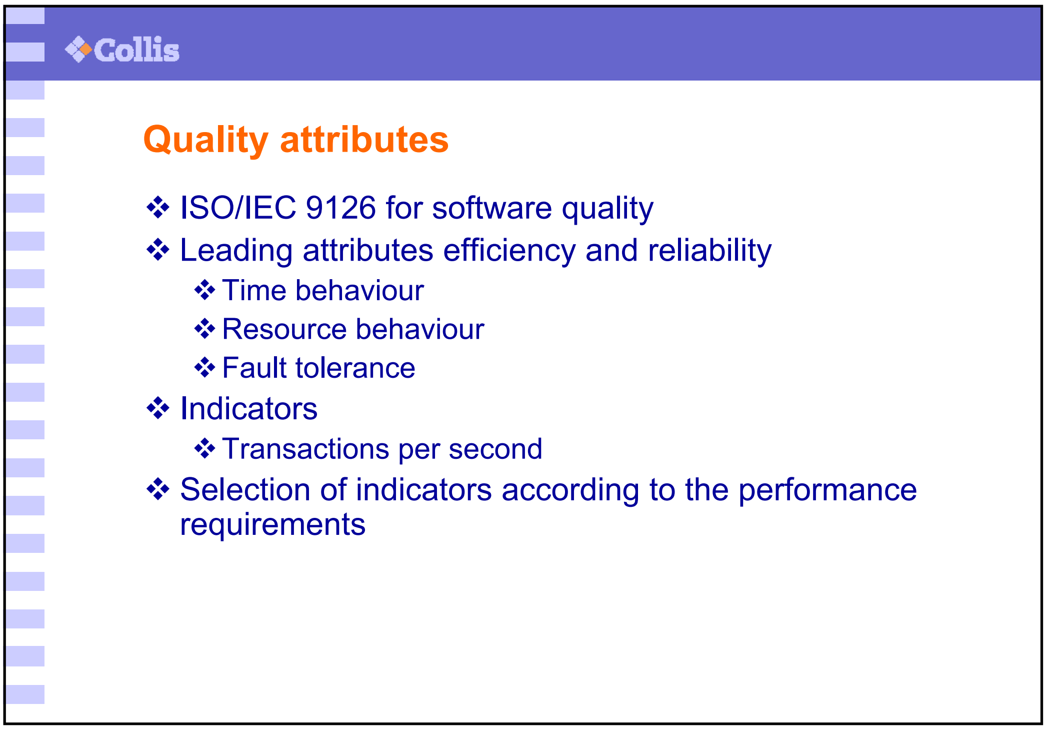
- It is hard to get some specific figures out of the performance test tools
- Performance measurements are the basis of representing the results
  - Measurement data has not been generalized so far
  - The tool manufacturers provide their own measurement data
- Need for a standard set of data on the basis of analyzing the results for the performance tester
- Comparison of test results



Collis

Performance test tools

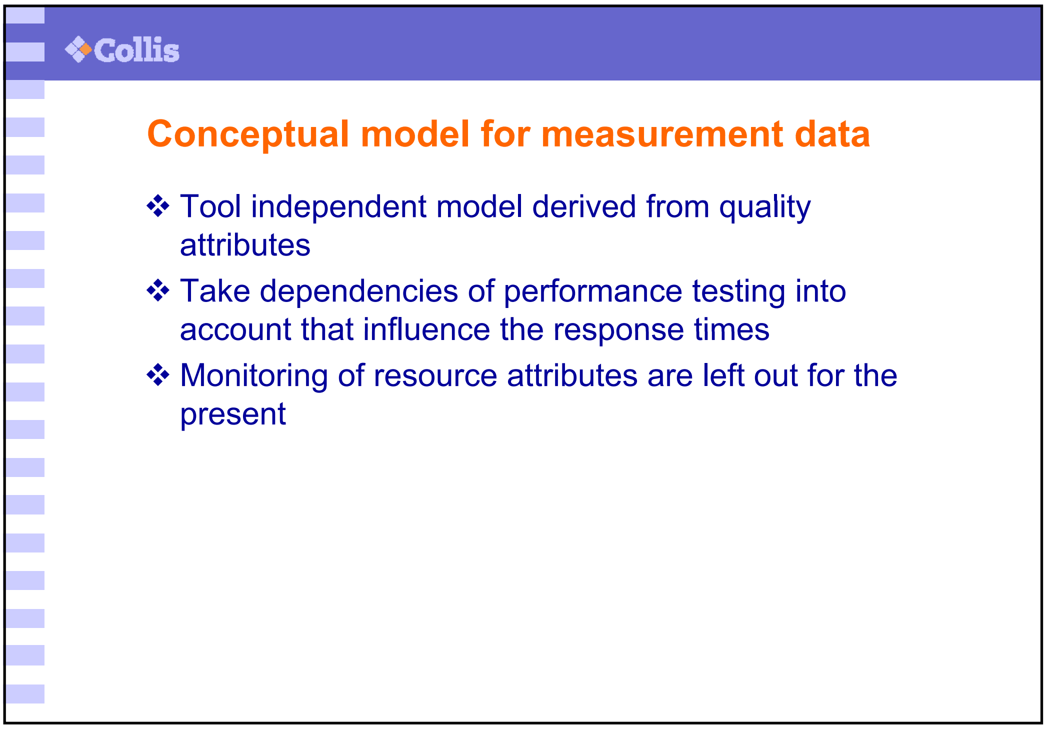
- Commercial tools like
  - Mercury Loadrunner
  - IBM Rational
  - Borland Silkperformer
- Freeware tools like:
  - The Grinder
  - OpenSTA
- Tools perform their own measurement data
- Most of the tools provide their own specific figures and their own graphs
- Finding: differences in handling stress situations between the tools



Collis

Quality attributes

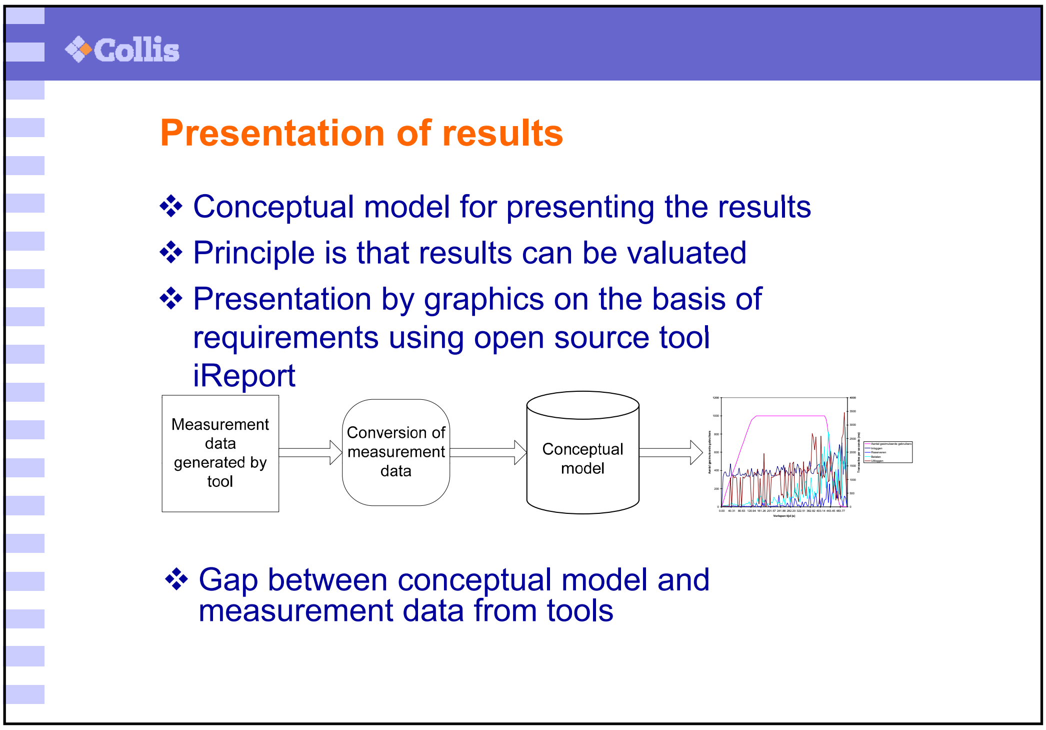
- ISO/IEC 9126 for software quality
- Leading attributes efficiency and reliability
  - Time behaviour
  - Resource behaviour
  - Fault tolerance
- Indicators
  - Transactions per second
- Selection of indicators according to the performance requirements



Collis

Conceptual model for measurement data

- Tool independent model derived from quality attributes
- Take dependencies of performance testing into account that influence the response times
- Monitoring of resource attributes are left out for the present



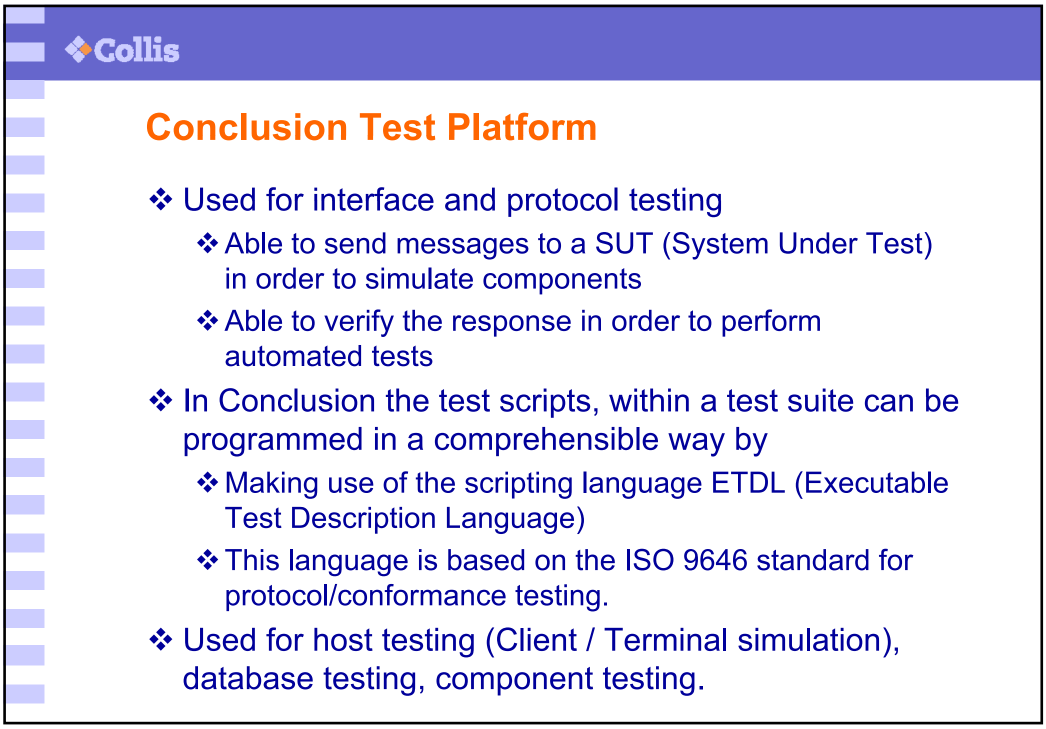
Collis

Presentation of results

- Conceptual model for presenting the results
- Principle is that results can be valuated
- Presentation by graphics on the basis of requirements using open source tool iReport

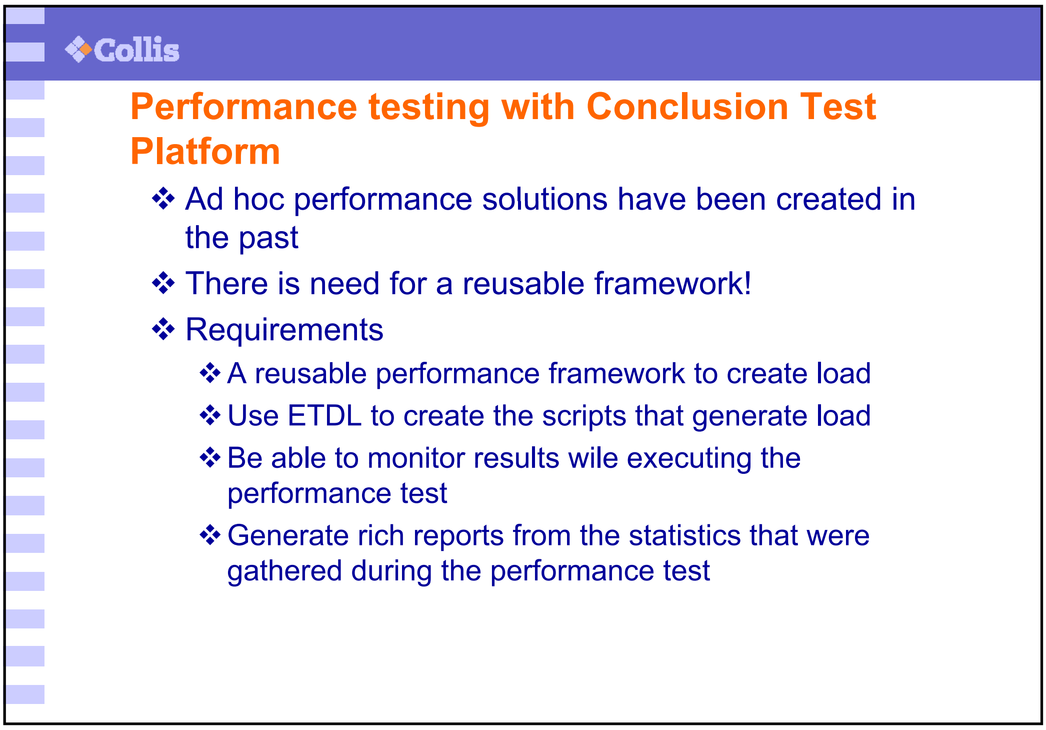
Measurement data generated by tool
Conversion of measurement data
Conceptual model

- Gap between conceptual model and measurement data from tools



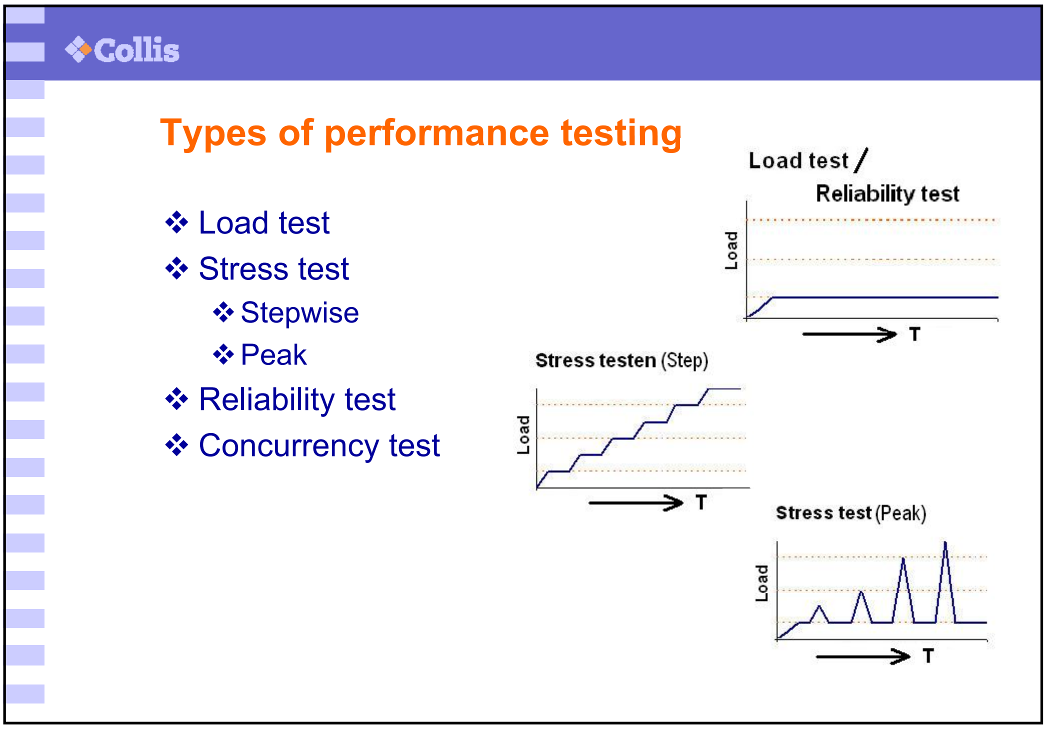
Conclusion Test Platform

- Used for interface and protocol testing
  - Able to send messages to a SUT (System Under Test) in order to simulate components
  - Able to verify the response in order to perform automated tests
- In Conclusion the test scripts, within a test suite can be programmed in a comprehensible way by
  - Making use of the scripting language ETDL (Executable Test Description Language)
  - This language is based on the ISO 9646 standard for protocol/conformance testing.
- Used for host testing (Client / Terminal simulation), database testing, component testing.



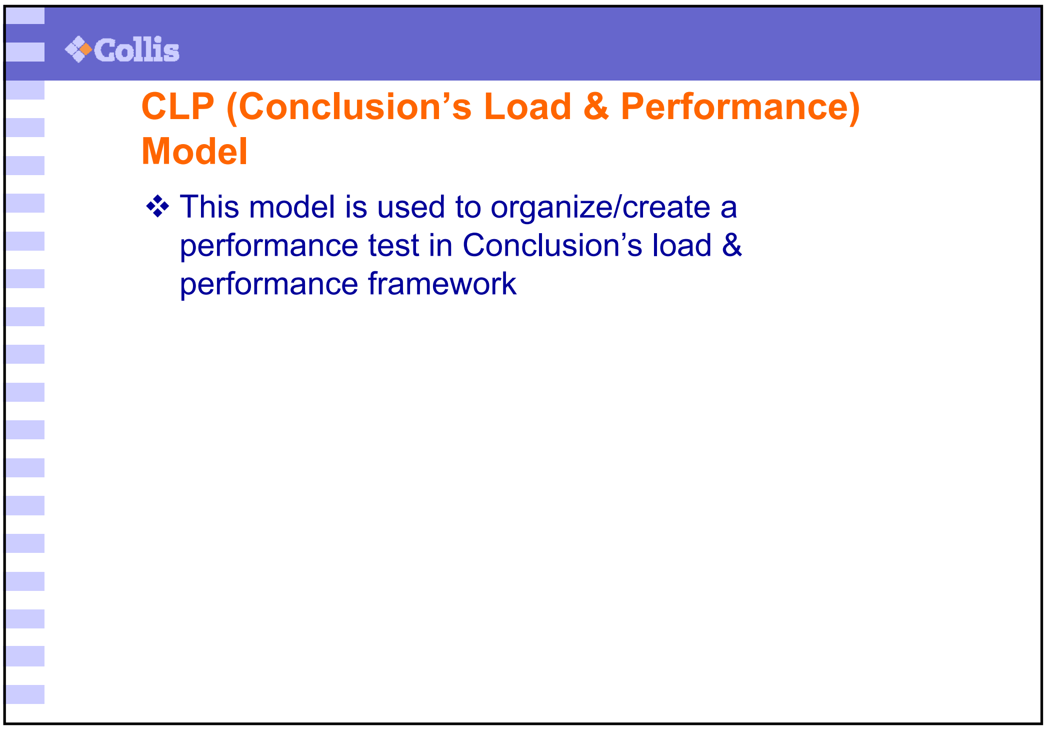
Performance testing with Conclusion Test Platform

- Ad hoc performance solutions have been created in the past
- There is need for a reusable framework!
- Requirements
  - A reusable performance framework to create load
  - Use ETDL to create the scripts that generate load
  - Be able to monitor results wile executing the performance test
  - Generate rich reports from the statistics that were gathered during the performance test



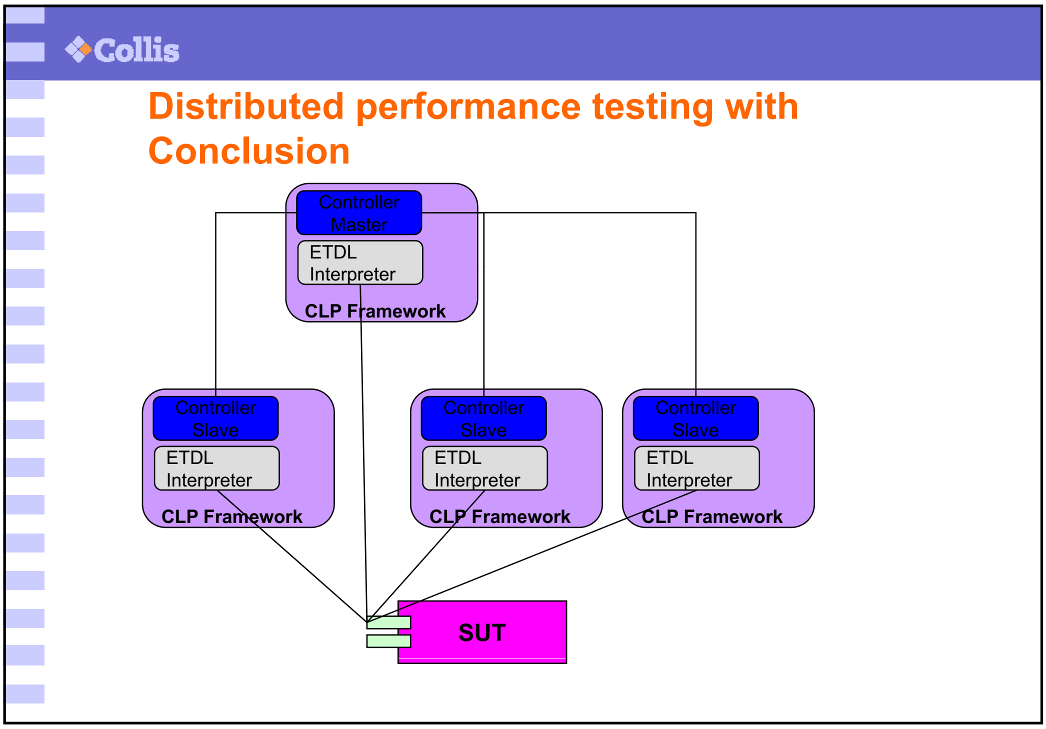
Types of performance testing

- Load test
- Stress test
  - Stepwise
  - Peak
- Reliability test
- Concurrency test



CLP (Conclusion's Load & Performance) Model

- This model is used to organize/create a performance test in Conclusion's load & performance framework



Collis

Distributed performance testing with Conclusion

Controller Master

ETDL Interpreter

CLP Framework

Controller Slave

ETDL Interpreter

CLP Framework

Controller Slave

ETDL Interpreter

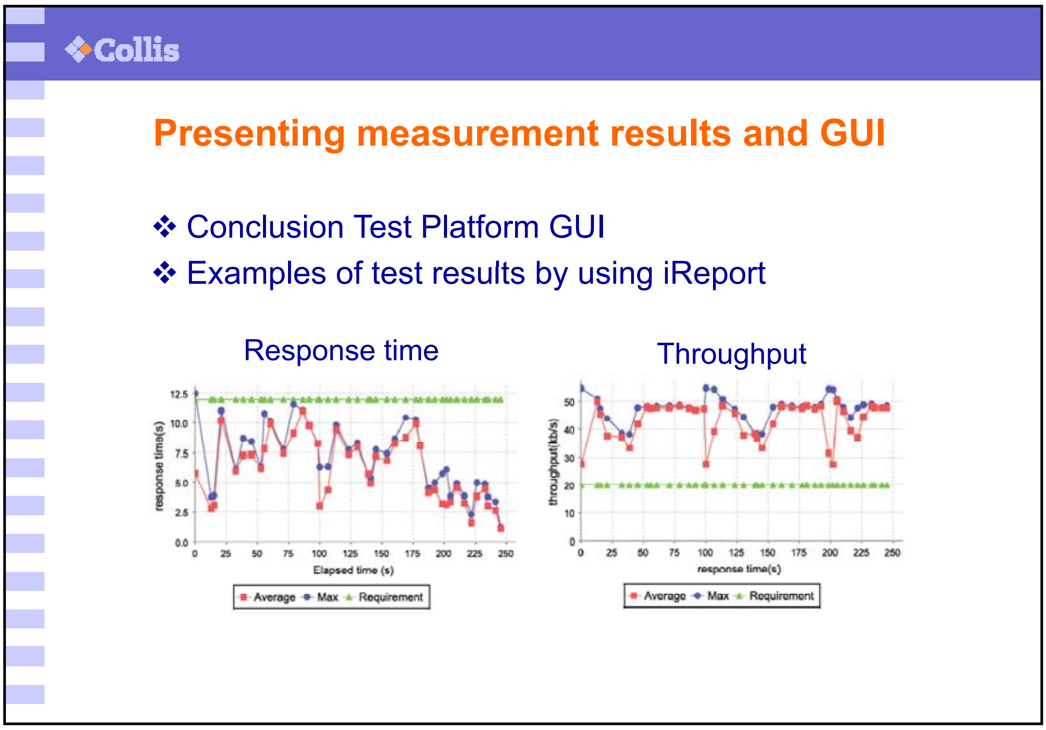
CLP Framework

Controller Slave

ETDL Interpreter

CLP Framework

SUT



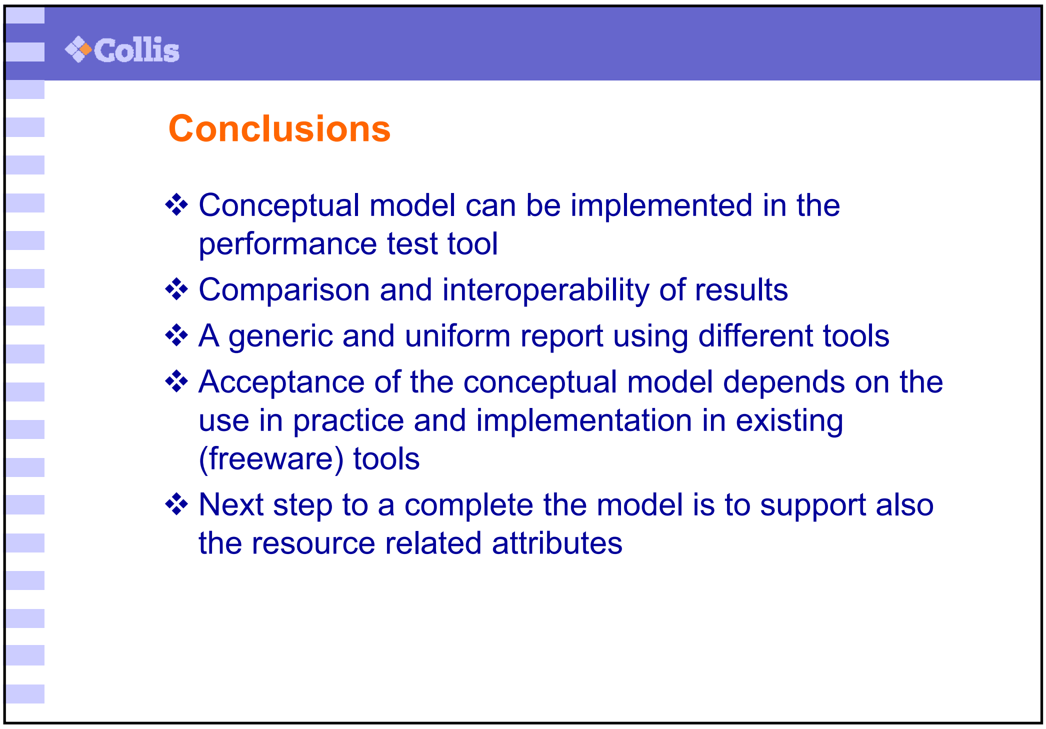
Collis

Presenting measurement results and GUI

- ❖ Conclusion Test Platform GUI
- ❖ Examples of test results by using iReport

Response time

Throughput



Collis

Conclusions

- Conceptual model can be implemented in the performance test tool
- Comparison and interoperability of results
- A generic and uniform report using different tools
- Acceptance of the conceptual model depends on the use in practice and implementation in existing (freeware) tools
- Next step to a complete the model is to support also the resource related attributes

WHITE PAPER

Managing .NET performance across the application life cycle

Software applications run the enterprise. In order to take advantage of new technology-enabled business models and processes, enterprise applications are becoming increasingly complex and distributed, and more frequently built on Microsoft's .NET architecture. Such applications have many moving parts, as they make use of numerous interconnected technologies spanning multiple computer systems.

Managing the performance of enterprise .NET applications in a live production environment has become a difficult problem that challenges IT organizations' traditional tools and processes. This paper will discuss the benefits of managing enterprise .NET application performance as a process that spans the application life cycle. It will also illustrate the use of Compuware software products that address domain-specific performance issues to facilitate this process.

The enterprise application performance imperative

Enterprise software applications are strategic assets that have become essential to the competitive operation of any organization. With the ongoing automation of most customer-facing and internal operations, a company's business competence is increasingly judged by the service provided by its applications. When those applications perform poorly or become unavailable, it has a negative and immediate impact on business operations and a longer-term effect on company image and reputation.

How does this happen? Poor application performance degrades the user experience. Lack of application availability eliminates the user experience altogether. If these users are external, such as customers or business partners, this should be a cause for great concern since customers are only a click away from the competition. The business suffers lost transactions and revenue, customer frustration and a poor reputation for availability and online presence. Poorly performing enterprise applications can have a similar effect on internal users, reducing employee productivity as the business slows to an uncompetitive pace. If internal users are customer-facing, these problems can further result in a diminished customer experience.

> **"Organizations deploying web services environments do so with the expectation that integrated systems will result in distributed execution platforms that will collectively support a given business service."[1]**
>
> –Enterprise Management Associates (EMA)

Needless to say, business reliance on live applications makes it imperative that software performance and availability problems are resolved more quickly than ever before. But why do performance problems happen in the first place? Why do applications that are tuned satisfactorily in development develop performance problems later in the application life cycle? Why is it so difficult for IT operations to solve .NET application performance problems in production? These are all valid questions that will be explored in the next several pages.

[1] "Reducing the Risks of Managing Web Services Environments," a Compuware-commissioned EMA white paper, August 2006.

Organizational challenges of sustaining application performance

Enterprise applications have grown up over the years. Mainframe and monolithic applications gave way to client-server architecture, which later evolved into three-tier distributed architecture. With the more recent integration of Internet-based technologies into mainstream three-tier business applications, we now have highly distributed, multi-tiered, Internet-enabled enterprise applications. The number of enabling technologies used in enterprise applications has increased exponentially over the past 10 years.

During this time developers, QA testers and IT operations analysts have adapted their practices to accommodate newer technologies, albeit within the context of their traditional IT silos. From the .NET perspective, development now creates distributed applications with .NET technology; QA tests these applications with .NET testing tools; yet IT operations monitor these applications with traditional device-centric monitoring tools. This puts IT operations at a disadvantage when dealing with .NET applications, mainly due to the lack of visibility into the Common Language Runtime (CLR). Unlike native applications written in C/C++, Visual Basic, COBOL or Fortran, .NET applications are hosted in a CLR, a runtime container that is at the heart of all .NET application servers. The CLR appears as a “black box” to traditional IT monitoring tools, which makes diagnosing .NET applications issues problematic for IT staff.

The resultant communication barrier between IT silos becomes painfully obvious when a .NET application performance problem arises in production. IT operations may be alerted to a performance problem through the help desk or an alert from a monitoring agent. Those responsible diligently check network and server status, CPU and memory utilization as well as other components of the infrastructure to ensure everything is normal. If the IT infrastructure is operating within acceptable service levels, the application itself or its runtime environment are blamed. This is mainly due to the average IT analyst’s lack of understanding of the .NET application or how to diagnose a performance problem that has no infrastructure symptoms. In the absence of any expedient options to remedy the problem, the IT analyst might choose to restart the application, reboot the application server environment or reboot the entire server in an effort to “clear” the problem. Unfortunately, this is only a temporary fix as the problem will recur periodically with no apparent cause.

Eventually a triage meeting convenes in which stakeholders attempt to isolate the problem to a particular device or infrastructure component. Since no single IT staff member has the tools or ability to troubleshoot a multi-tiered .NET application performance problem, the natural tendency is for each member to demonstrate that his or her part of the infrastructure is performing properly. The servers are healthy, the network is fine, the database is okay, and therefore the problem must be with the application. Yet, developers and testers counter with evidence the application has passed functional and load testing, and has performed just fine in production until now.

Triage sessions like this occur in corporate IT departments around the globe on a regular basis. They are unproductive because every stakeholder views the problem through the lens of their own domain-specific tools and knowledge. The problem is not a deficiency in any one area, but the lack of collaboration between stakeholders combined with a lack of planning for a .NET application triage scenario in production. Creating and sustaining application performance and availability in today’s complex distributed computing environments calls for a life-cycle process to manage application performance, and the proper tools to solve problems quickly through collaboration across IT disciplines.

93 percent of surveyed developers and operations staff said it was either challenging or very challenging to quickly identify and resolve .NET performance problems.[2]

[2] “Go Speed Racer, Go: Make Your .NET Applications Run Mach 5!” Compuware webcast, August 2006.

The application performance life cycle

To develop, deploy and maintain high-performing applications, organizations must integrate performance into the application life cycle. This requires enhancing software engineering and IT practices to include specific performance-related activities at each appropriate stage of the life cycle, and using the right tools to facilitate those activities.

Performance-related, life-cycle activities should include:

- performance objectives and requirements definition
- architecture and design reviews for performance
- performance testing, tuning and optimization in development
- test case and test suite timing analysis in QA
- pre-production load testing and performance base-lining
- application service-level specification and monitoring
- production-level application performance management.

In practice, the activities are more numerous and detailed, but the following list should provide a reasonable high-level understanding of the performance life-cycle discussion that follows.

Life-cycle Phase	Performance-related Activities	Compuware Solutions
Requirements Definition	Business requirements Technical performance requirements	Optimal Trace
Architecture and Design	Architecture definition Design prototypes Prototype performance analysis	DevPartner® Studio
Development	Code performance analysis Memory utilization analysis Code optimization Performance design review(s)	DevPartner® Studio
Quality Assurance	Functional testing Test-suite performance metrics Performance requirements validation QA performance baselining	*QA*Center Enterprise Edition DevPartner® Studio
Pre-production Deployment	Load and stress testing Application optimization Performance baselining Predictive analysis	*QA*Center Performance Edition Vantage
Production	SLA monitoring End-user experience monitoring Network and server monitoring Production-level performance analysis SLA compliance reporting	Vantage

Planning

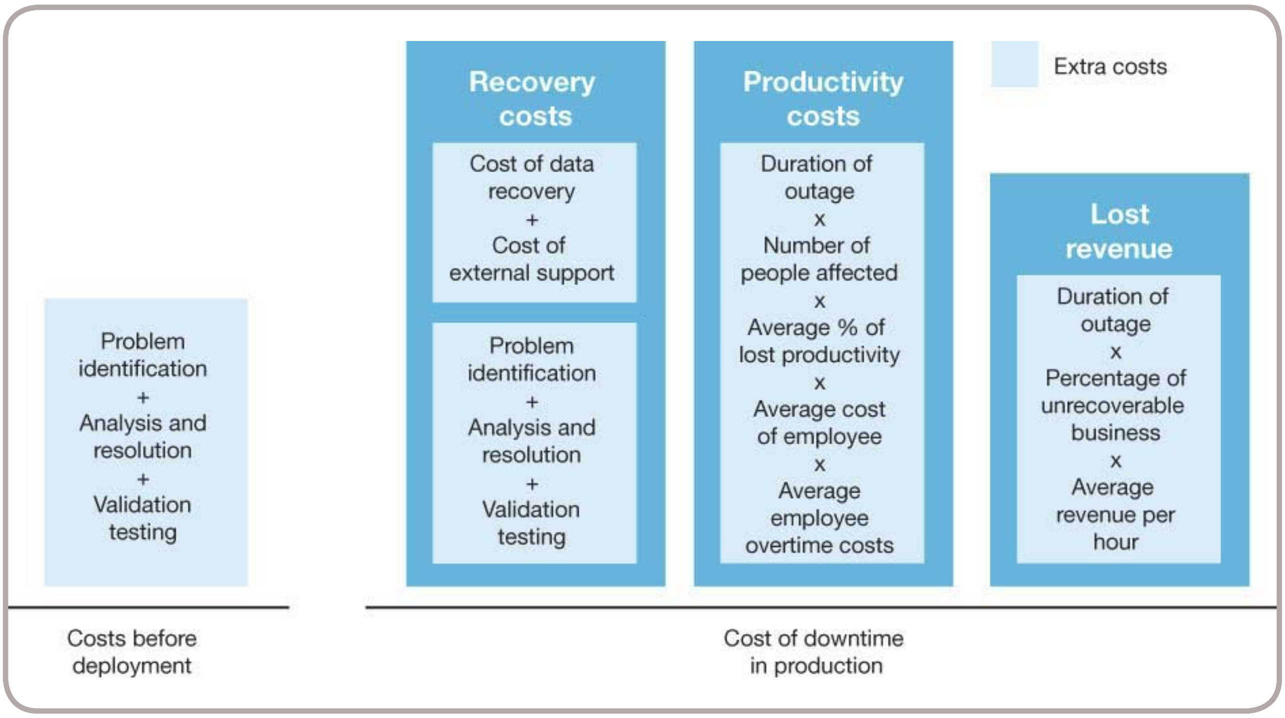
During the planning phase, line-of-business stakeholders normally define application needs and objectives in terms of business processes and functions using non-technical terminology. While performance is rarely a consideration at this stage, it is important that business planners and analysts state their expectations with high-level performance and capacity objectives.

Such objectives may sound something like this: “maximum response time of two seconds for a customer lookup transaction”; “must be able to support 1,200 internal users and up to 30,000 external users simultaneously”; or “must be able to process 85,000 point-of-sale transactions per minute from 2,100 retail locations.” Though coarse and non-technical in nature, these objectives will serve as a starting point for the requirements and analysis phase of the application life cycle. They will also be used to determine parameters for application load testing and production service-level agreements.

Requirements definition

Well-defined requirements are critical because they drive all subsequent phases of a project. Whether you are building a house, a road or a .NET application, requirements dictate the desired end result. With a house you might specify the number of floors, rooms and windows while with a software application, well-defined performance requirements should specify throughput, response times, scalability and so on.

At this stage, the business unit’s coarse performance objectives must be translated into more granular technical requirements. Since the objectives are almost always unclear and incomplete, software product managers must clarify their intended meaning by collaborating with line-of-business planners. While this can be a tedious exercise in patience and communication, it is a critical step that must be executed correctly to ensure technical requirements will deliver the results needed to satisfy the intent of the business objectives.


Extra costs
Problem identification + Analysis and resolution + Validation testing
Recovery costs
Cost of data recovery + Cost of external support
Problem identification + Analysis and resolution + Validation testing
Productivity costs
Duration of outage x Number of people affected x Average % of lost productivity x Average cost of employee x Average employee overtime costs
Lost revenue
Duration of outage x Percentage of unrecoverable business x Average revenue per hour
Costs before deployment
Cost of downtime in production

The cost of problem resolution is far higher when projects are in production.
Performance Management And The Application Life Cycle**,** Forrester Research, Inc., February 2005.

When defining performance-related application requirements, it is important that no implementation assumptions are made. Good requirements specify the desired result in accurate high-level technical terms without limiting the implementation options needed by architecture and design. Legitimate exceptions to this rule would include limitations passed along from the business unit, such as compatibility with existing hardware or software.

Clear, formal and measurable performance requirements ensure the business unit's performance objectives are considered throughout the rest of the life cycle. If the software product requirements process is informal, performance issues are often overlooked until the last possible moment, which is far too late. Solving application performance problems in production is very expensive as application outages and brownouts translate into lost revenue and productivity costs (see illustration on page 4).

Architecture and design

During the architecture and design phase, technical performance requirements are further decomposed into elements of a design proposal. For example, if a three-tier architecture is proposed, the requirement for a two-second customer lookup transaction must be broken down into design criteria for the presentation layer, business logic and database access. As distributed application environments become more complex, the performance overhead imposed by network latency, proxy servers and multiple layers of security must also be considered. In aggregate, the per-tier technical performance of the proposed design, plus infrastructure latency, should be comfortably within the stated technical performance requirements and business objectives.

To meet these requirements, it may be necessary to create more than one design prototype and measure the performance of each. Compuware DevPartner Studio is well-suited for this task, with the ability to collect detailed timing information across multiple tiers of a .NET application. The results are correlated in a multi-tier view that quickly identifies and categorizes the most CPU and memory-intensive classes, methods and lines of code. By comparing the performance metrics of two prototype implementations, architects and designers can make design decisions based on accurate timing data.

Architecture and design staff must remain engaged in implementation decisions that could affect performance as the application progresses through development. At least one performance design review should be conducted prior to feature completion on smaller applications, while more complex applications will require several reviews. With developers also using DevPartner Studio during development and unit testing, the performance design review is accelerated by the availability of performance metrics generated automatically during development.

Development

While good architecture and design create a foundation for good application code, developers actually implement the application features and functionality as specified in the product requirements and design specifications. The knowledge and expertise of the software development team is arguably one of the most critical determinants of an effective implementation. At this stage in the life cycle, bad things can happen to good applications.

Poor coding techniques can easily and transparently introduce performance-robbing side effects into the application code base. A poorly coded application can easily deliver 100 percent of required features along with hidden performance bottlenecks, memory utilization problems and potentially fatal thread synchronization issues. Such problems may even slip through QA testing unnoticed, until they are exposed in load testing or live production. By that time, the cost and business impact of finding and fixing problems in the application code is significantly higher than if they were corrected in development.

In the absence of any specific performance requirements, testing will focus almost exclusively on functionality needed to meet requirements and pass QA standards. However, with clearly defined performance requirements, developers must be cognizant of code performance issues to deliver an implementation that meets those requirements. If they fail to do so, QA test results should expose the problems using performance metrics collected during automated functional testing.

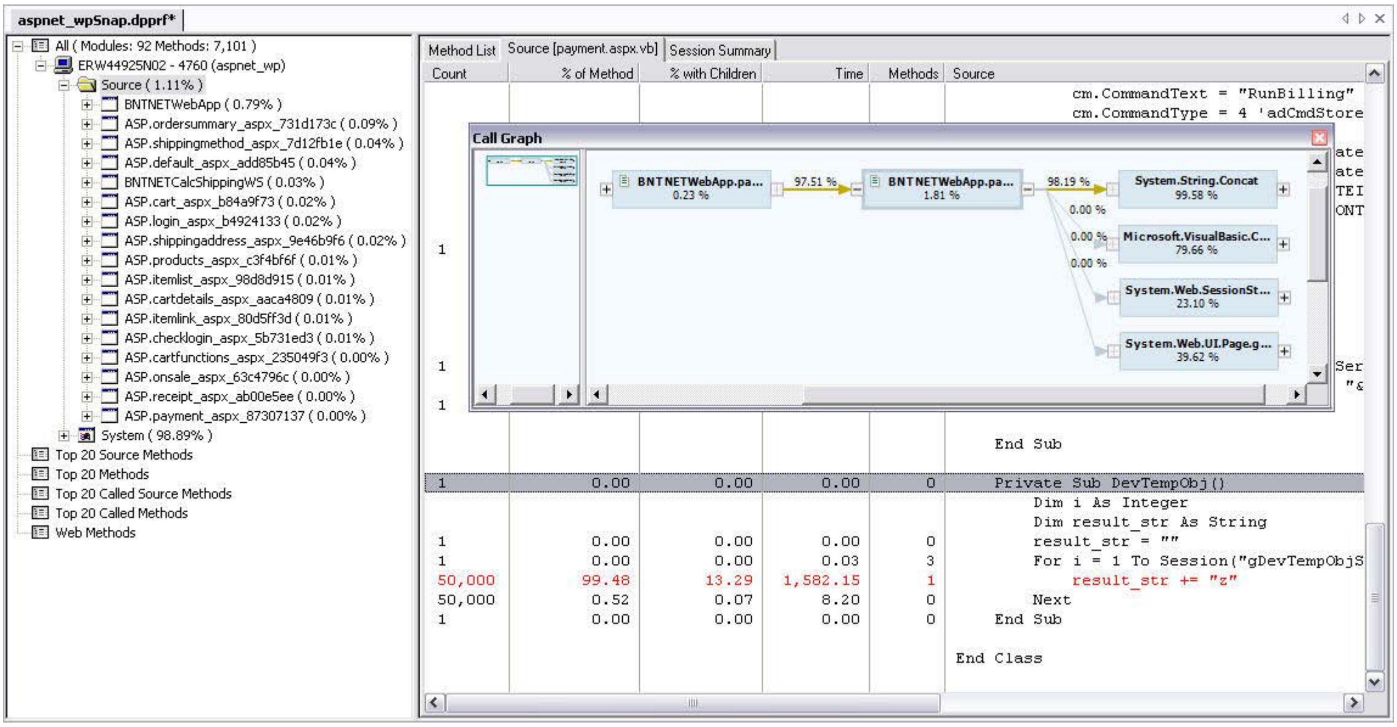
Performance tuning an application is a good development practice for ensuring code executes quickly with no significant bottlenecks. Similarly, memory profiling an application in development is effective for ensuring correct and efficient use of memory resources. Thread synchronization analysis is a third development-specific task that can help optimize runtime performance and avoid potentially fatal thread deadlocks, starvation and race conditions.

In general, developers should consider each and every optimization has a cumulative effect on improving overall application performance. In daily practice, developers must remember that any piece of functionality can be coded in two or more different ways. By following the architecture and design practice of considering at least two prototypes on a regular basis, developers can profile multiple prototypes to identify the best-performing option at the earliest possible stage. Many performance problems are suitable for correction in development, some in QA functional testing and others in pre-production load testing. When development optimization is neglected, problems that could have been resolved early on become more difficult and more expensive to correct. These problems can further complicate the resolution of load-related performance issues that may be discovered later in the pre-production environment.

Performance analysis and optimization tools, such as those found in Compuware DevPartner Studio, are effective because they provide developers with metrics to base good optimization decisions earlier in the application life cycle. That's not to say development code optimizations are adequate for ensuring acceptable and sustained performance throughout the life cycle. On the contrary, development optimization is only one phase in the performance life cycle, albeit a very important one that saves time, effort, expense and unnecessary delays later.

Platform and configuration differences between development, QA, pre-production and deployment environments dictate the type of tuning effort that is appropriate at each stage. Without a reasonable set of guidelines, you'll spend too much time tuning and optimizing during the development cycle. In a three-tier .NET application, all of the presentation-layer components can and should be tuned with precision by the time the code is feature-complete. At this stage, it is appropriate to optimize any rich-client .NET code, Active Server Pages (ASPXs), web services and browser-hosted scripting. Applications that are exposed as web services need additional optimization during development, particularly in the marshalling, un-marshalling and transformation of XML.

In general, applications will perform differently in a production environment than in a development/test environment, due to platform architecture and configuration differences. As a result, performing only coarse-level tuning of these components during development is more efficient. Further optimization of middle-tier and back-end components is more precise when code performance and resource utilization data is collected in a pre-production environment that closely resembles the target production environment.



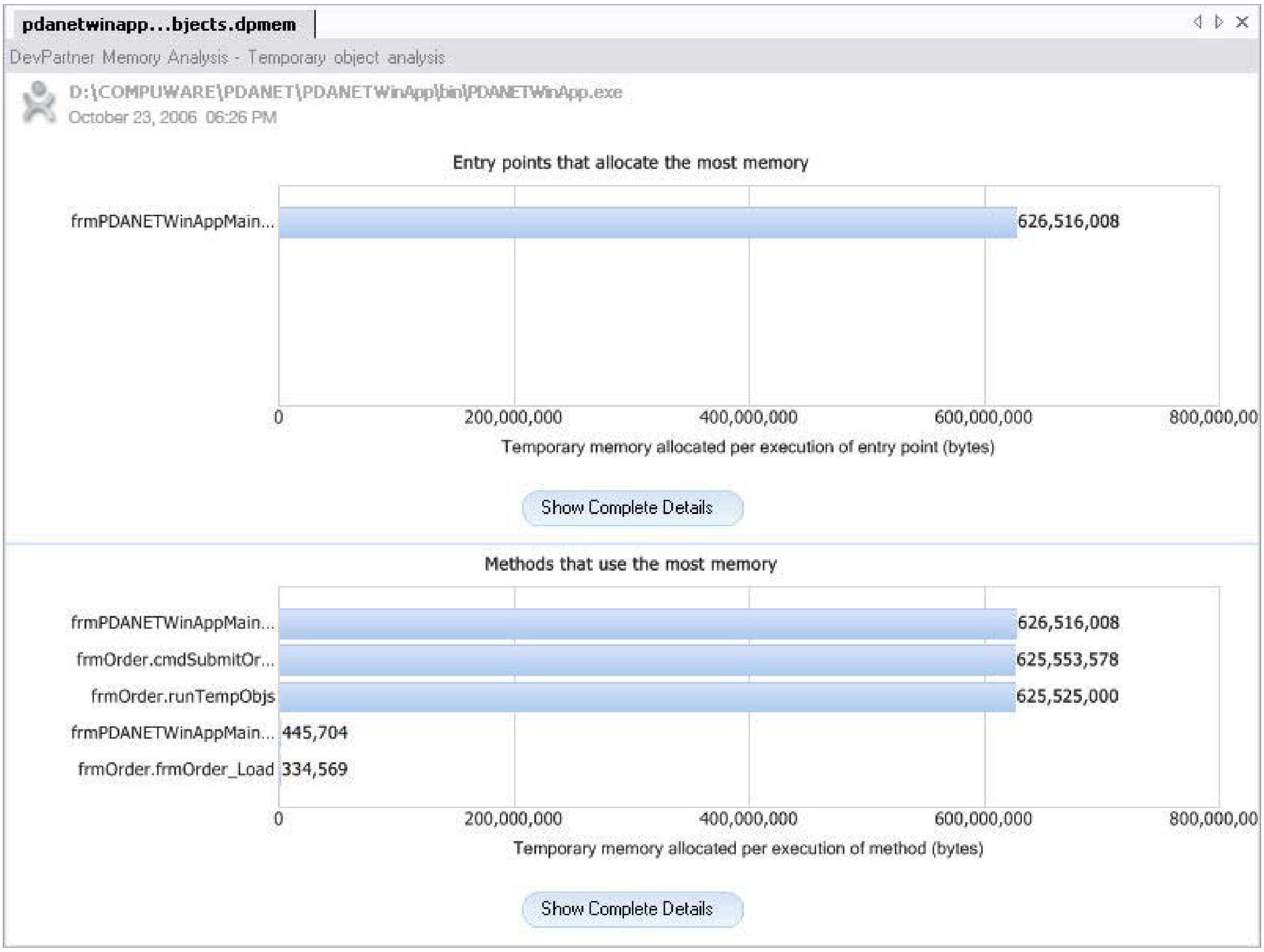
aspnet_wpSnap.dpprf*

- All (Modules: 92 Methods: 7,101)
  - ERW44925N02 - 4760 (aspnet_wp)
    - Source (1.11%)
      - BNTNETWebApp (0.79%)
      - ASP.ordersummary_aspx_731d173c (0.09%)
      - ASP.shippingmethod_aspx_7d12fb1e (0.04%)
      - ASP.default_aspx_add85b45 (0.04%)
      - BNTNETCalcShippingWS (0.03%)
      - ASP.cart_aspx_b84a9f73 (0.02%)
      - ASP.login_aspx_b4924133 (0.02%)
      - ASP.shippingaddress_aspx_9e46b9f6 (0.02%)
      - ASP.products_aspx_c3f4bf6f (0.01%)
      - ASP.itemlist_aspx_98d8d915 (0.01%)
      - ASP.cartdetails_aspx_aaca4809 (0.01%)
      - ASP.itemlink_aspx_80d5ff3d (0.01%)
      - ASP.checklogin_aspx_5b731ed3 (0.01%)
      - ASP.cartfunctions_aspx_235049f3 (0.00%)
      - ASP.onsale_aspx_63c4796c (0.00%)
      - ASP.receipt_aspx_ab00e5ee (0.00%)
      - ASP.payment_aspx_87307137 (0.00%)
    - System (98.89%)
  - Top 20 Source Methods
  - Top 20 Methods
  - Top 20 Called Source Methods
  - Top 20 Called Methods
  - Web Methods

Method List
Source [payment.aspx.vb]
Session Summary
Call Graph
BNTNETWebApp.pa... 0.23 %
97.51 %
BNTNETWebApp.pa... 1.81 %
98.19 %
System.String.Concat 99.58 %
0.00 %
0.00 %
Microsoft.VisualBasic.C... 79.66 %
0.00 %
System.Web.SessionSt... 23.10 %
System.Web.UI.Page.g... 39.62 %

```
Count        % of Method   % with Children        Time   Methods   Source
                                                                           cm.CommandText = "RunBilling"
                                                                           cm.CommandType = 4 'adCmdStore
                                                                                                      ate
                                                                                                      ate
                                                                                                      TEI
                                                                                                      ONT
1
1                                                                                                     Ser
1                                                                                                      "&
                                                                     End Sub

1                   0.00          0.00          0.00         0           Private Sub DevTempObj()
                                                                             Dim i As Integer
                                                                             Dim result_str As String
1                   0.00          0.00          0.00         0               result_str = ""
1                   0.00          0.00          0.03         3               For i = 1 To Session("gDevTempObjS
50,000             99.48         13.29      1,582.15         1                   result_str += "z"
50,000              0.52          0.07          8.20         0               Next
1                   0.00          0.00          0.00         0           End Sub

                                                                     End Class
```

Figure 1: By utilizing the performance analysis feature of DevPartner Studio, developers can see which tiers, methods and lines of code that have the most impact on response time.



pdanetwinapp...bjects.dpmem

DevPartner Memory Analysis - Temporary object analysis

D:\COMPUWARE\PDANET\PDANETWinApp\bin\PDANETWinApp.exe
October 23, 2006 06:26 PM

Figure 2: DevPartner Studio highlights the methods that use the most short/medium/long-lived temporary objects and provides guidance for resource optimization when profiling memory allocations and de-allocations.

Quality assurance

QA testers focus primarily on functional testing to ensure application features and functions meet requirements. Whether manual or automated testing is performed, the outcome of each functional test is a pass/fail result indicating whether the code responded as the tester or test script expected. While an opportunity exists to measure performance during functional testing, particularly in automated testing environments, it is usually overlooked and deferred until after the QA cycle. Seizing the opportunity to identify latent performance issues during functional testing can avoid the higher costs associated with finding problems later in pre-production or live production environments.

With clearly defined performance requirements, QA testers can develop a performance test plan and associated test suite to ensure the application meets those requirements. In theory, code performance can be measured on all tests and test suites beginning early in the QA cycle. In practice, however, this would produce such a high volume of detailed timing data that its value would diminish due to the difficulty of identifying meaningful data. The data could easily be ignored altogether.

One way to avoid this problem is to map each performance requirement to the functional tests that exercise code for the specified feature, operation or transaction. These tests should be considered on the performance-critical path. Whenever these performance-relevant tests are run, performance data should be automatically collected and compared to the previous run(s) and to timing thresholds derived from performance requirements. The comparison should yield a performance pass/fail result based on requirement thresholds, a faster/slower performance result when compared to previous test runs and a separate pass/fail result for the functional test.

In an automated test environment such as Compuware *QA*Center, each automated functional test is timed from start to finish. This wall-clock timing approach is adequate for QA test assessment and for functional tests that are not on the performance critical path. Unfortunately it is far too coarse and unreliable for measuring code performance on the critical path. The same test could be run several times producing different wall-clock results for each run due to system loading, network traffic and other environmental factors. Wall-clock timing can easily generate false or misleading results if system loading or other environmental factors cause a performance-related test to execute slowly.

Further complicating this issue is the fact that a small 100-line .NET program can execute thousands, or even tens of thousands, of lines of library and system code during a test. When a functional test executes slowly based on wall-clock timing, there is no effective means of identifying which class or method of application code is responsible for the problem. Coarse-level wall-clock timing is simply inadequate for communicating meaningful information back to developers. It makes troubleshooting and correcting problems more difficult and time-consuming in development, that is, without any specific information on which class, method or line of code was responsible for the problem.

A proper solution for accurate timing of code under test is Compuware DevPartner Studio. Its performance analysis feature can be used in conjunction with automated testing solutions like *QA*Center to produce highly accurate and repeatable timing data. DevPartner Studio performance analysis measures CPU cycles and execution time with precise granularity, attributing relevant timing data to each appropriate line of .NET code, method or class in the application. It also measures separately the time a .NET application spends executing library code, common language runtime code and underlying system code. By gathering detailed metrics and excluding CPU cycles used by code that is not on the performance critical path, DevPartner Studio is an ideal product to use in conjunction with *QA*Center automated testing. When QA performance metrics indicate that a test has executed slower than a previous run, or below threshold requirements, the DevPartner Studio session file containing detailed timing metrics will quickly reveal the precise location of the problem to QA testers and developers, thus reducing the time to resolution. Once an application has passed QA functional testing, the final set of performance session files should be archived as the performance baseline for the application in the QA functional test environment. A separate performance baseline will be captured during load testing in a pre-production environment.

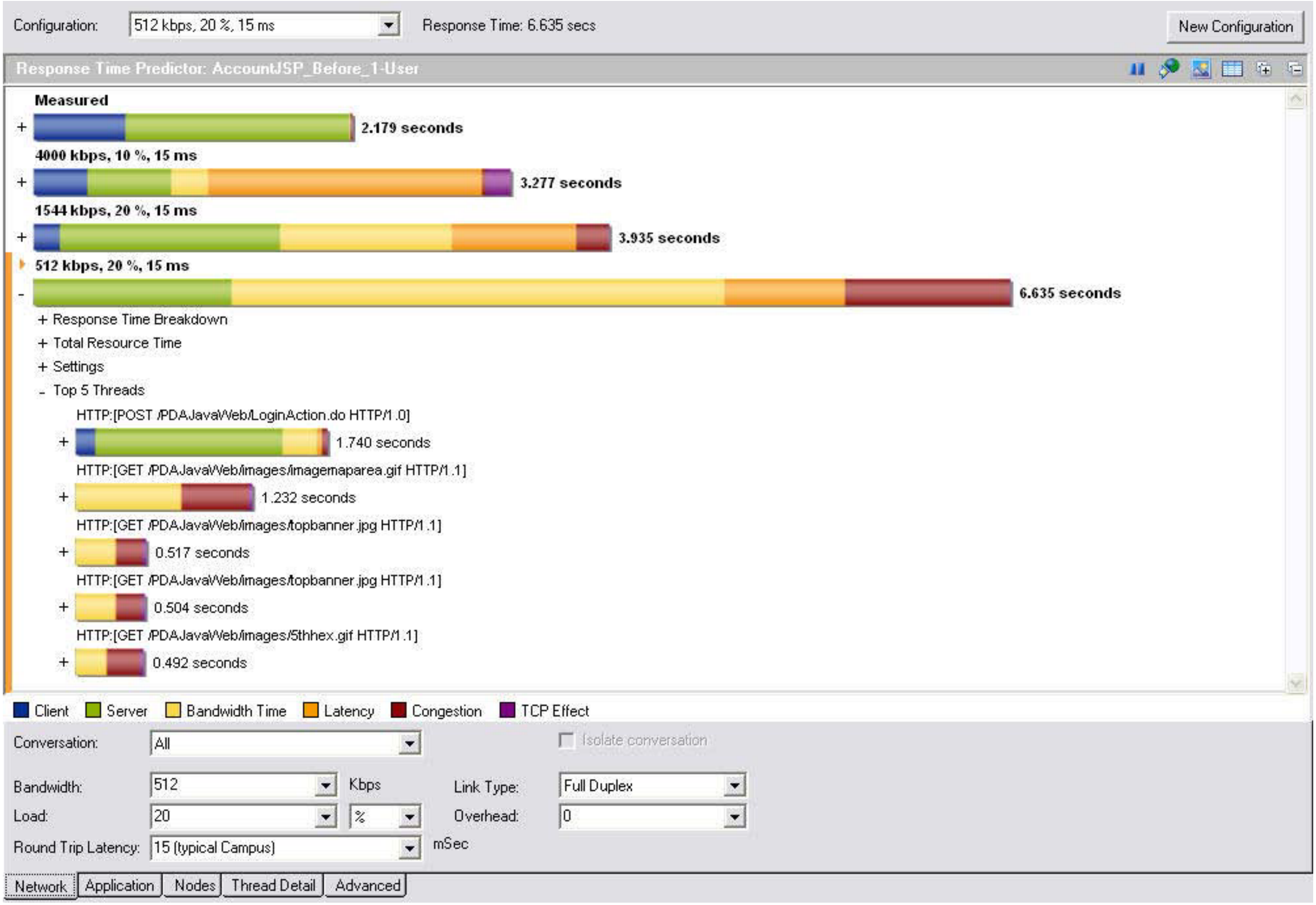
Pre-production performance assurance

Once an application has reached an acceptable level of quality, it is typically moved to a pre-production environment that more closely resembles the production environment. At this stage, IT analysts normally determine system, network and other resource demands the application will make when it is moved to the live production environment. While IT analysts are very concerned with the impact this new application will have on the production environment and its inclusive resources, they are not particularly concerned about the performance of the application code itself. This is precisely where the interests of development, QA and IT operations begin to diverge.

The pre-production stage is likely the first time in the application life cycle that the code will execute in an environment similar to the target production environment. Load testing a feature-complete application in a pre-production environment is one of the best predictors of real-world application performance, and has become a common practice for delivering enterprise-class business applications. This environment is suited to test and measure an enterprise application's ability to scale to peak user loads while maintaining required service levels.

Before full-scale load testing can be performed, a trial load test with a small number of users exercising key transactions will normally reveal an obvious scalability problem somewhere in the application. Since a .NET application is distributed over multiple nodes and network segments, technicians must somehow isolate which part of the application or infrastructure is responsible for the problem. The diagnostic work and resources required to isolate such problems can become very complex and often unwieldy. Compuware Vantage offers a solution to this problem.

Vantage is an effective solution for isolating and troubleshooting application transaction performance problems in pre-production and production environments. It identifies the causes of poor end-user response times wherever they reside—on client workstations, the



Configuration: 512 kbps, 20 %, 15 ms

Response Time: 6.635 secs

Response Time Predictor: AccountJSP_Before_1-User

Measured

+ 2.179 seconds

4000 kbps, 10 %, 15 ms

+ 3.277 seconds

1544 kbps, 20 %, 15 ms

+ 3.935 seconds

512 kbps, 20 %, 15 ms

- 6.635 seconds

+ Response Time Breakdown

+ Total Resource Time

+ Settings

- Top 5 Threads

HTTP:[POST /PDAJavaWeb/LoginAction.do HTTP/1.0]

+ 1.740 seconds

HTTP:[GET /PDAJavaWeb/images/imagemaparea.gif HTTP/1.1]

+ 1.232 seconds

HTTP:[GET /PDAJavaWeb/images/topbanner.jpg HTTP/1.1]

+ 0.517 seconds

HTTP:[GET /PDAJavaWeb/images/topbanner.jpg HTTP/1.1]

+ 0.504 seconds

HTTP:[GET /PDAJavaWeb/images/5thhex.gif HTTP/1.1]

+ 0.492 seconds

🟦 Client 🟩 Server 🟨 Bandwidth Time 🟧 Latency 🟥 Congestion 🟪 TCP Effect

Conversation: All

□ Isolate conversation

Bandwidth: 512 Kbps

Link Type: Full Duplex

Load: 20 %

Overhead: 0

Round Trip Latency: 15 (typical Campus) mSec

Figure 3: With Vantage, you can see the response time of a transaction in various networked environment configurations and adjust numerous characteristics (such as bandwidth, latency and load) to predict what will happen in a production environment.

network, servers or even in the application itself—thus eliminating time-consuming guesswork. Vantage also provides a predictive troubleshooting component that allows network administrators to see the impact of various performance adjustments on transaction response time, such as:

- modifying bandwidth, latency and load
- increasing/decreasing the power of a server (see below)
- changing application turns for individual “thread” components, varying TCP window size.

Once the coarse-level scalability problems are resolved, the application is ready for load testing with a much larger number of users. Like functional testing, load testing can involve a manual or automated testing methodology. As one might readily conclude, manual load testing calls for a vast number of client computers along with the appropriate number of human testers needed to exercise the application by following a manual test script. To the vast numbers of enterprise application users, manual load testing is normally prohibitive in terms of time, resources and potential for error. Surprisingly, some applications are still tested this way.

On the other hand, automated load testing software products are able to simulate hundreds or thousands of virtual users on a single computer. Each virtual user exercises the application using automated test scripts. Automated load testing tools not only require far fewer humans than manual testing, but also provide the opportunity to stress-test an application by scaling the number of virtual users to a level beyond anticipated use. The result is a very powerful testing solution that no enterprise software organization should be without.

Compuware *QA*Center Performance Edition provides accurate and scalable load testing by emulating hundreds or thousands of simultaneous virtual users. It can run large, accurate and repeatable load tests using only minimal hardware resources. With its powerful data management tool, *QA*Center Performance Edition creates valid and accurate test data that is representative of real-world use, further ensuring load test results will accurately predict application performance and availability in production.

When an application performance problem is uncovered this late in the application life cycle, it is important to execute an expedient resolution to keep the project on schedule. The challenge, then, is to enable pre-production IT staff to collect data that is suitable for developers to analyze and fix the problem quickly. Although performance analysis tools like DevPartner Studio can collect this detailed data, their use in a production or pre-production environment is not appropriate for IT analysts. IT analysts need the capability to collect detailed performance data for the developers, but with a feature set and user interface geared to the IT operations professional.

Compuware Vantage is the appropriate solution. Vantage makes it easy for IT operations staff to zero in on .NET problems without requiring a high level of .NET expertise. It provides actionable insight into enterprise .NET application performance problems, facilitates better communication between IT silos and helps identify the responsible stakeholders quickly.

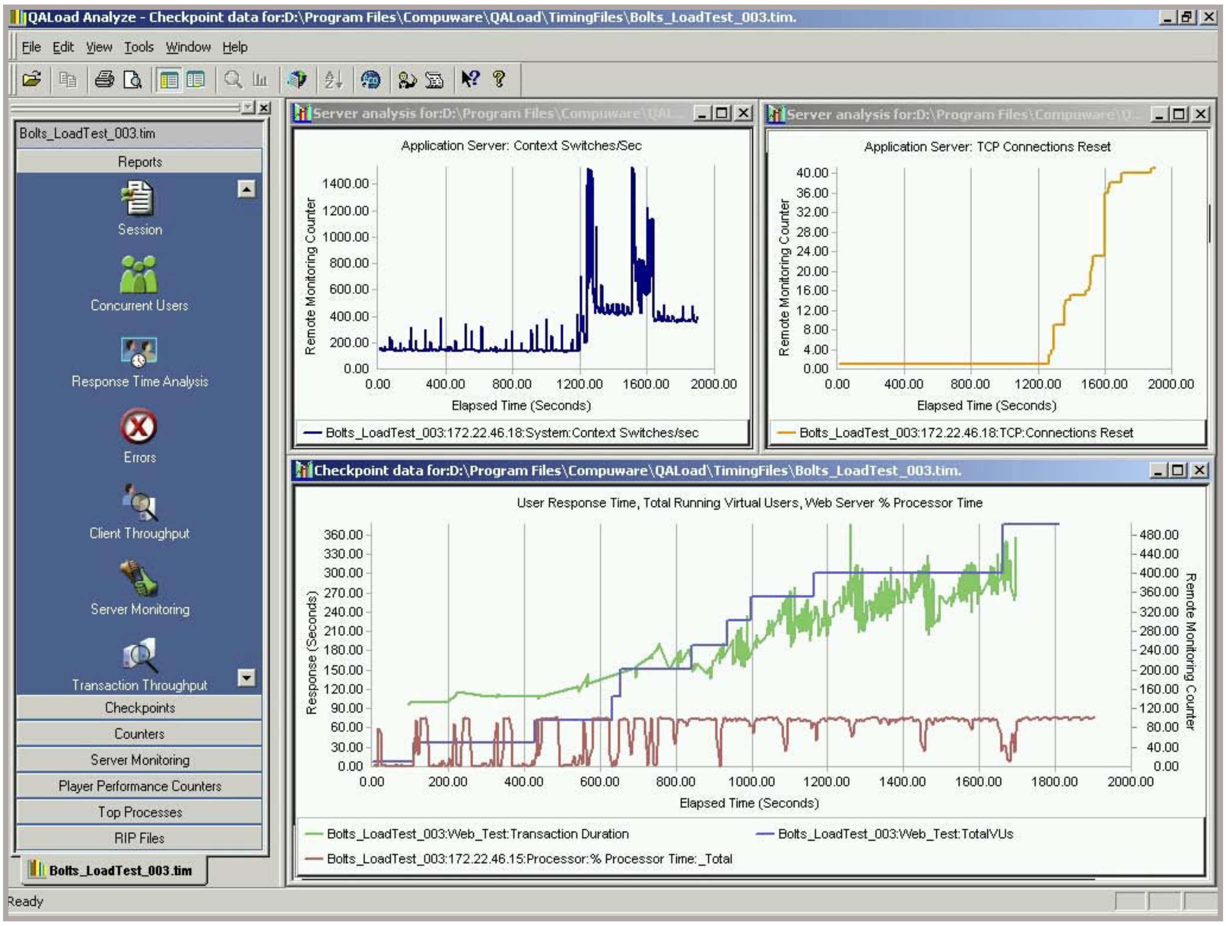
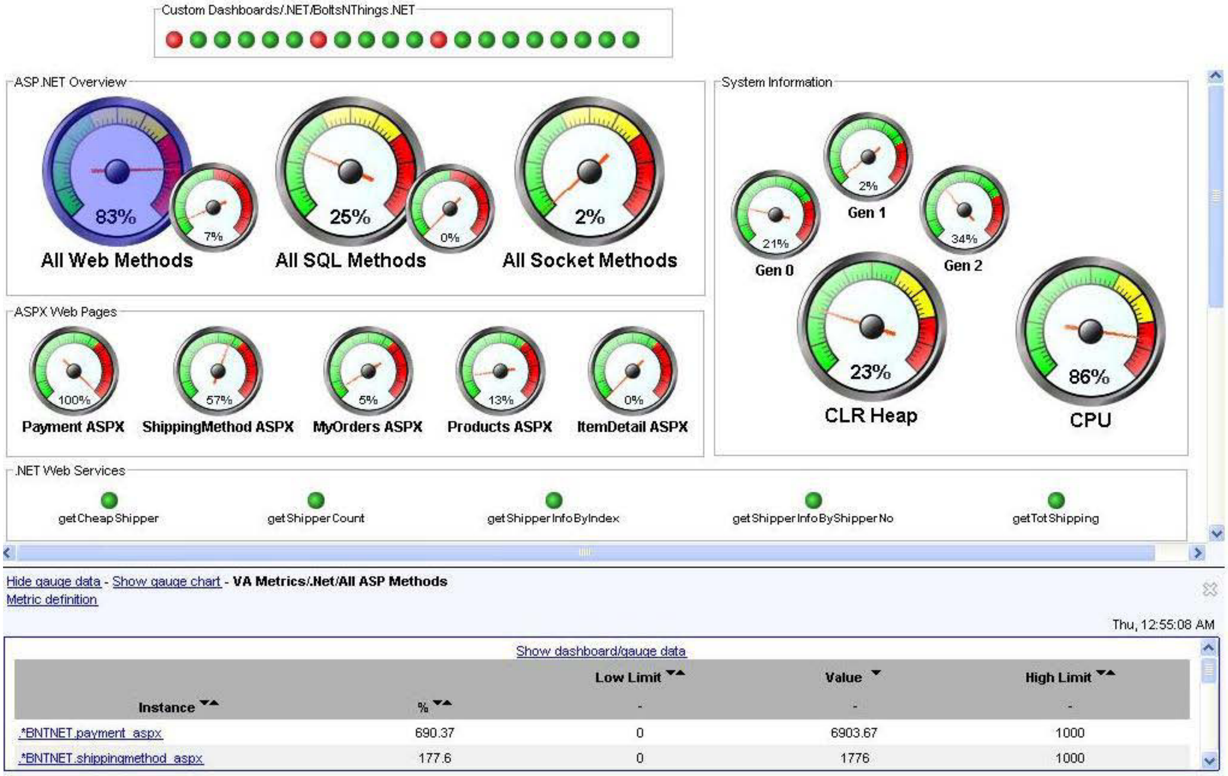

QALoad Analyze - Checkpoint data for:D:\Program Files\Compuware\QALoad\TimingFiles\Bolts_LoadTest_003.tim.
File
Edit
View
Tools
Window
Help
Bolts_LoadTest_003.tim
Reports
Session
Concurrent Users
Response Time Analysis
Errors
Client Throughput
Server Monitoring
Transaction Throughput
Checkpoints
Counters
Server Monitoring
Player Performance Counters
Top Processes
RIP Files
Bolts_LoadTest_003.tim
Ready
Server analysis for:D:\Program Files\Compuware\QAL...
Application Server: Context Switches/Sec
Remote Monitoring Counter
1400.00
1200.00
1000.00
800.00
600.00
400.00
200.00
0.00
0.00
400.00
800.00
1200.00
1600.00
2000.00
Elapsed Time (Seconds)
Bolts_LoadTest_003:172.22.46.18:System:Context Switches/sec
Server analysis for:D:\Program Files\Compuware\Q...
Application Server: TCP Connections Reset
Remote Monitoring Counter
40.00
36.00
32.00
28.00
24.00
20.00
16.00
12.00
8.00
4.00
0.00
0.00
400.00
800.00
1200.00
1600.00
2000.00
Elapsed Time (Seconds)
Bolts_LoadTest_003:172.22.46.18:TCP:Connections Reset
Checkpoint data for:D:\Program Files\Compuware\QALoad\TimingFiles\Bolts_LoadTest_003.tim.
User Response Time, Total Running Virtual Users, Web Server % Processor Time
Response (Seconds)
360.00
330.00
300.00
270.00
240.00
210.00
180.00
150.00
120.00
90.00
60.00
30.00
0.00
Remote Monitoring Counter
480.00
440.00
400.00
360.00
320.00
280.00
240.00
200.00
160.00
120.00
80.00
40.00
0.00
0.00
200.00
400.00
600.00
800.00
1000.00
1200.00
1400.00
1600.00
1800.00
2000.00
Elapsed Time (Seconds)
Bolts_LoadTest_003:Web_Test:Transaction Duration
Bolts_LoadTest_003:Web_Test:TotalVUs
Bolts_LoadTest_003:172.22.46.15:Processor:% Processor Time:_Total

Figure 4: *QA*Center Performance Edition has the flexibility to modify a test while it runs, allowing testers to add and remove users to speed up problem resolution.



VA Metrics/.Net/All ASP Methods

Thu, 12:55:08 AM

		Low Limit	Value	High Limit
Instance	%	-	-	-
.*BNTNET.payment_aspx	690.37	0	6903.67	1000
.*BNTNET.shippingmethod_aspx	177.6	0	1776	1000

Figure 5: The Vantage Web Services Dashboard provides a user-customizable interface to monitor the overall health of an application. It can be made up of one or more gauges based on user-selected metrics, such as WMI and/or method-level data.

Vantage provides analysis capabilities that go right to the source of specific problems such as CPU/Wait Time-intensive program code and transactions, as well as long-running SQL statements, thus exposing the root cause of .NET application performance, availability and scalability problems. When used in conjunction with *QA*Center Performance Edition during pre-production load testing, Vantage captures performance data on .NET components that can lead developers quickly and easily to the source of the problem.

Once an application has been optimized to pass load testing requirements, the final set of load test and performance session files should be archived as the performance baseline for the application as measured in the pre-production environment.

Production deployment

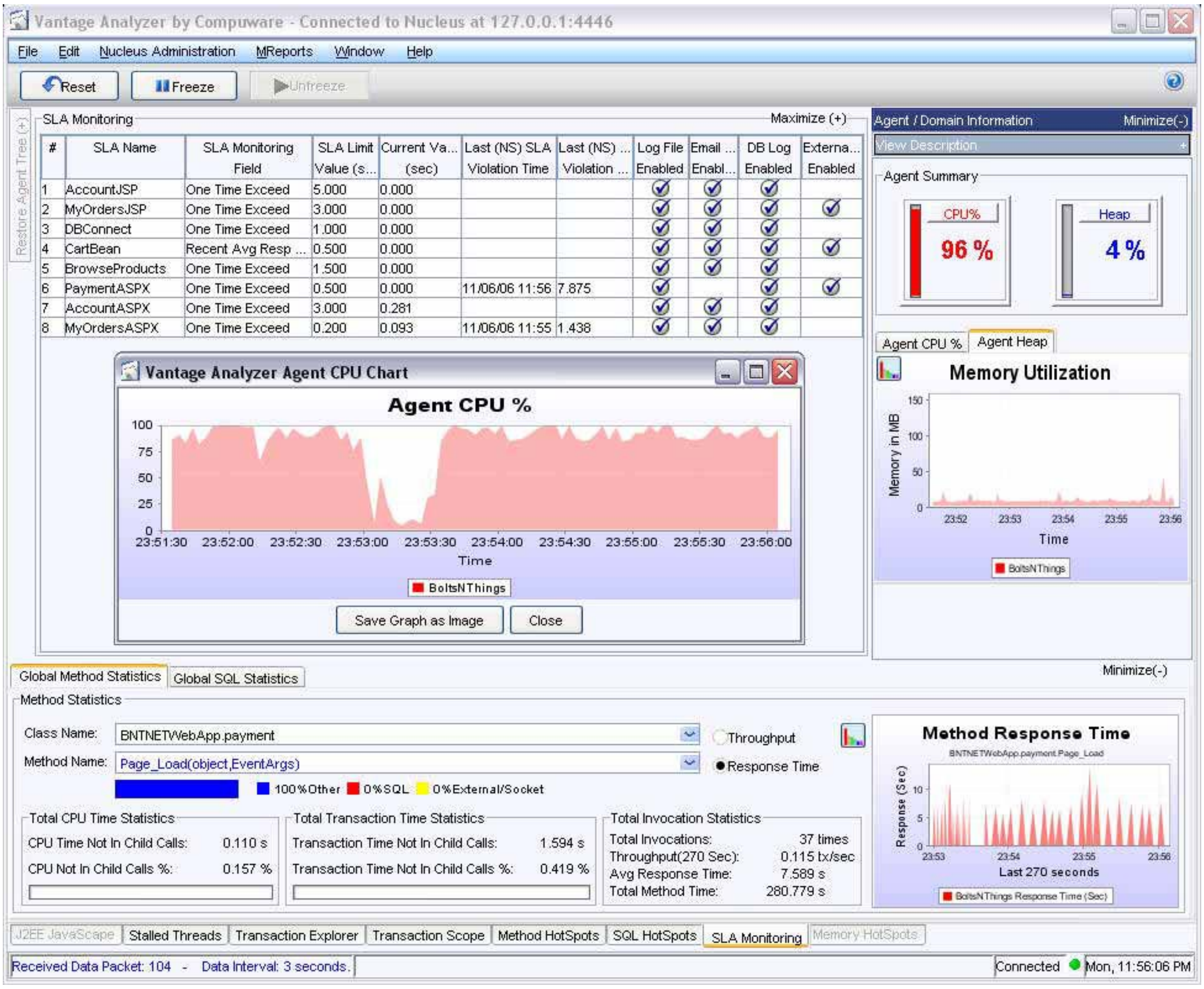
Once the application is deployed in a production environment, it is subjected to environmental forces that cannot be controlled or anticipated during development, QA or pre-production. The effects of system-level and infrastructure changes, application server updates and other environment changes made by peer applications all add up to a dynamically changing runtime environment that becomes less representative of the pre-production environment over time.

Deploying an enterprise application today also requires meeting service level agreements (SLAs). While a newly deployed application may indeed meet SLAs comfortably, there is no guarantee that it will continue to do so over time. Myriad factors in the ever-changing production environment can contribute to changes in application performance.

Performance problems that arise in production can negatively impact the business immediately, and need to be resolved quickly and effectively. Unfortunately, once an application goes live, it is in the domain of IT operations, out of the reach of developers and their application-centric tools. Network analysts, systems analysts and other support staff monitor the infrastructure in which the application runs, and possibly even the application server container in which it runs, but have little knowledge about the health of the .NET application itself.

When a production application falls below required service levels, an alert may be triggered or calls may begin to flood the help desk. In either case, IT analysts take notice. They look at server, database and network utilization, but rarely come up with a root cause in short order. Once they figure out everything seems to be “performing normally” in the infrastructure, the development organization is consulted. While operations can confirm an application has slowed to a crawl, there is usually very little useful information they can offer developers.

Vantage is ideally suited to bridging the gap between IT operations and development in application triage scenarios like the one we’ve just described. Vantage provides continuous monitoring of .NET application performance and resource utilization without impacting production-level performance. It monitors and reports on SLA compliance, and captures detailed and actionable data at the moment an SLA threshold is exceeded. When IT operation staff members respond to an alert from Vantage, they can quickly review the nature of the SLA violation and navigate to the root cause of the problem in a .NET application. If the problem requires the involvement of development, Vantage session data will provide the information developers need in .NET terminology they understand, and in the context of their application and source code.



#	SLA Name	SLA Monitoring Field	SLA Limit Value (s...	Current Va... (sec)	Last (NS) SLA Violation Time	Last (NS) ... Violation ...	Log File Enabled	Email ... Enabl...	DB Log Enabled	Externa... Enabled
1	AccountJSP	One Time Exceed	5.000	0.000			✓	✓	✓	
2	MyOrdersJSP	One Time Exceed	3.000	0.000			✓	✓	✓	✓
3	DBConnect	One Time Exceed	1.000	0.000			✓	✓	✓	
4	CartBean	Recent Avg Resp ...	0.500	0.000			✓	✓	✓	✓
5	BrowseProducts	One Time Exceed	1.500	0.000			✓	✓	✓	
6	PaymentASPX	One Time Exceed	0.500	0.000	11/06/06 11:56	7.875	✓		✓	✓
7	AccountASPX	One Time Exceed	3.000	0.281			✓	✓	✓	
8	MyOrdersASPX	One Time Exceed	0.200	0.093	11/06/06 11:55	1.438	✓	✓	✓	

Class Name: BNTNETWebApp.payment

Method Name: Page_Load(object,EventArgs)

Total CPU Time Statistics

CPU Time Not In Child Calls:	0.110 s
CPU Not In Child Calls %:	0.157 %

Total Transaction Time Statistics

Transaction Time Not In Child Calls:	1.594 s
Transaction Time Not In Child Calls %:	0.419 %

Total Invocation Statistics

Total Invocations:	37 times
Throughput(270 Sec):	0.115 tx/sec
Avg Response Time:	7.589 s
Total Method Time:	280.779 s

Received Data Packet: 104 - Data Interval: 3 seconds.

Figure 6: The Vantage SLA view can be configured to trigger alarms, such as sending a page/e-mail or run a command, to alert administrators of degraded performance, resource starvation, hung transactions, system response-time issues or the approach of system transaction capacity.

In addition to improving communication between IT operations and development on .NET issues, Vantage can also help IT operations communicate valuable transaction data back to QA. QA functional and load tests normally exercise a wide range of application features, functions and transactions with equal distribution. With production-level data on transaction counts and .NET component utilization from Vantage, QA groups can improve test effectiveness by modifying test parameters to more closely resemble real-world scenarios.

Often, the 80/20 rule applies to software utilization: 20 percent of the code executes 80 percent of the time. Armed with specific information on which .NET components make up the “hot” 20 percent, .NET developers can make incremental optimizations on that code while QA can improve test suites to identify how these optimizations will improve performance under real-world utilization scenarios. While a 0.0001 second optimization may seem insignificant by itself, when multiplied by the frequency of real-world execution, which can easily be millions of times per day, the cumulative performance is both significant and measurable.

Once an application or portfolio of applications has been deployed with acceptable performance results, ongoing monitoring and management of applications and infrastructure is an important factor for documenting SLA compliance and ensuring sustained application performance.

In the earlier section “Organizational challenges of sustaining application performance,” we explored the gaps in traditional IT monitoring tools in the context of diagnosing .NET application performance problems. Many organizations measure service only at the device level, leaving performance issues undetected until users begin to complain. When service delivery is approached as an integrated whole, IT organizations must manage both infrastructure performance, as well as application service-levels, to meet business demands and priorities. To accomplish this, IT organizations need proper tools to:

- measure end-user experience of application availability and response times
- prioritize performance issues based on their business impact
- systematically resolve problems via accurate analysis of performance issues
- monitor and analyze application performance automatically and continuously across the client, network, server and database tiers
- capture performance analysis information at the time of service-level exceptions
- integrate all monitoring and measurements into useful, management-level reports that expedite service and support, and reduce confusion and delay.

Vantage integrates deep performance analysis with robust end-user experience monitoring, providing the ability to follow a transaction over the network and into the data center. When the end-user performance of an application degrades, the Vantage CNS (Client, Network, Server) Exception report provides a top-down, end-to-end view of the poorly performing transaction. With a glance, it’s easy to determine whether a performance problem resides with the client, network or server environment. When the CNS Exception report indicates a server problem, technical staff can drill down quickly into a Vantage View to determine whether a slow ASPX page is application-intensive, or if it is waiting on back-end processing such as SQL or third-party calls. Having this unique visibility provides a method by which IT staff can agree where performance problems reside and makes assigning and resolving the problem a cinch.

> **“Businesses struggling to stabilize support costs and protect themselves against the inherent risks of managing web services environments should definitely evaluate this solution (Vantage).”[3]**
>
> –Enterprise Management Associates (EMA)

Compuware Vantage delivers on the promise of comprehensive, integrated application service management. Vantage helps IT organizations manage application performance from the perspective that matters most: the end user. With response time metrics integrated with end-to-end performance analysis, Vantage enables IT managers to proactively identify and rapidly resolve tough performance problems. By managing applications at the business, transaction and infrastructure level, Vantage provides the critical insight needed to improve application service and maximize the value of application and infrastructure investments.

[3] “Reducing the Risks of Managing Web Services Environments,” a Compuware-commissioned EMA white paper, August 2006.

Conclusion

Creating and sustaining application performance and availability in today's complex distributed computing environments calls for a life-cycle process to ensure application performance, and the proper tools to solve problems quickly through collaboration across IT disciplines. Enterprise application performance planning must begin at the earliest possible stage, preferably at the business-unit level, even prior to technical requirements and analysis. Clearly defined performance requirements offer the best assurance application performance will be considered from architecture and design, through development and QA, and on into the production environment.

Software defects, both functional and performance-related, are extremely expensive to fix once an application has reached production. By adhering to best practices at each stage of the application life cycle, application performance can be optimized properly at each stage, avoiding costly downtime and application brownouts.

Compuware offers a wide range of products and services that help IT organizations worldwide meet application performance requirements throughout the application life cycle. Compuware DevPartner Studio helps developers and QA testers improve .NET application performance during the early stages of the application life cycle. Compuware Vantage helps pre-production QA testers and IT operations ensure .NET application performance during the latter stages of the application life cycle.

Vantage is an essential tool for IT organizations responsible for maintaining service level agreements on enterprise .NET applications. Vantage moves beyond the capability of traditional IT monitoring tools, giving IT organizations visibility into the Common Language Runtime "black box," and reports actionable data that expedites problem resolution while improving communication between development, QA and IT operations.

To learn more about Vantage, visit
www.compuware.com/vantage

Compuware Corporation Corporate Headquarters
One Campus Martius
Detroit, MI 48226

For regional and international office contacts, please visit our web site at www.compuware.com

COMPUWARE®
www.compuware.com

Fault Diagnosis of Embedded Software using Program Spectra*

Peter Zoeteweij[1] Rui Abreu[1] Rob Golsteijn[2] Arjan J.C. van Gemund[1]

[1]Embedded Software Lab,
Software Engineering Research Group, TU Delft
{p.zoeteweij,r.f.abreu,a.j.c.vangemund}@tudelft.nl

[2]NXP
rob.golsteijn@nxp.com

Abstract

Automated diagnosis of errors detected during software testing can improve the efficiency of the debugging process, and can thus help to make software more reliable. In this paper we discuss the application of a specific automated debugging technique, namely software fault localization through the analysis of program spectra, in the area of embedded software in high-volume consumer electronics products. We discuss why the technique is particularly well suited for this application domain, and through experiments on an industrial test case we demonstrate that it can lead to highly accurate diagnoses of realistic errors.

Keywords: diagnosis, program spectra, automated debugging, embedded systems, consumer electronics.

1 Introduction

Software reliability can generally be improved through extensive testing and debugging, but this is often in conflict with market conditions: software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. In this typical scenario, testing reveals more bugs than can be solved, and debugging is a bottleneck for improving reliability. Automated debugging techniques can help to reduce this bottleneck.

The subject of this paper is a particular automated debugging technique, namely software fault localization through the analysis of *program spectra* [11]. These can be seen as projections of execution traces that indicate which parts of a program were active during various runs of that program. The diagnosis consist in analyzing the extent to which the activity of specific parts correlates with errors detected in the different runs.

Locating a fault is an important step in actually solving it, and program spectra have successfully been applied for this purpose in several tools focusing on various application domains, such as Pinpoint [4], which focuses on large, dynamic on-line transaction processing systems, AMPLE [5], which focuses on object-oriented software, and Tarantula [9], which focuses on C programs.

In this paper, we discuss the applicability of the technique to embedded software, and specifically to embedded software in high-volume consumer electronics products. Software has become an important factor in the development, marketing, and user-perception of these products, and the typical combination of limited computing resources, complex systems, and tight development deadlines make the technique a particularly attractive means for improving product reliability.

To support our argument, we report the outcome of two experiments, where we diagnosed two different errors occurring in the control software of a particular product line of television sets from a well-known international consumer electronics manufacturer. In both experiments, the technique is able to locate the (known) faults that cause these errors quite well, and in one case, this implies an accuracy of a single statement in approximately 450K lines of code.

The remainder of this paper is organized as follows. In Section 2 we explain the diagnosis technique in more detail, and in Section 3 we discuss its applicability to embedded software in consumer electronics products. In Section 4 we describe our

*This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

experiments, and in Section 5 we discuss how our current implementation can be improved. In Section 6 we discuss related work. We conclude in Section 7.

2 Preliminaries

In this section we introduce program spectra, and describe how they are used for diagnosing software faults. First we introduce the necessary terminology.

2.1 Failures, Errors, and Faults

As defined in [3], we use the following terminology.

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is the part of the total state of the system that may cause a failure.
- A *fault* is the cause of an error in the system.

To illustrate these concepts, consider the C function in Figure 1. It is meant to sort, using the bubble sort algorithm, a sequence of `n` rational numbers whose numerators and denominators are passed via parameters `num` and `den`, respectively. There is a fault (bug) in the swapping code of block 4: only the numerators of the rational numbers are swapped. The denominators are left in their original order.

A failure occurs when applying `RationalSort` yields anything other than a sorted version of its input. An error occurs after the code inside the conditional statement is executed, while `den[j]` $\neq$ `den[j+1]`. Such errors can be temporary: if we apply `RationalSort` to the sequence $\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, an error occurs after the first two numerators are swapped. However, this error is "canceled" by later swapping actions, and the sequence ends up being sorted correctly. Faults do not automatically lead to errors either: no error will occur if the input is already sorted, or if all denominators are equal.

The purpose of *diagnosis* is to locate the faults that are the root cause of detected errors. As such, error detection is a prerequisite for diagnosis. As a rudimentary form of error detection, failure detection can be used, but in software more powerful mechanisms are available, such as pointer checking, array bounds checking, deadlock detection, etc.

In a software context, faults are often called *bugs*, and diagnosis is part of *debugging*. Computer-aided techniques as the one we consider here are known as *automated debugging*.

```
void RationalSort(int n, int *num, int *den
    )
{   /* block 1 */
    int i,j,temp;

    for ( i=n-1; i>=0; i-- ) {
        /* block 2 */
        for ( j=0; j<i; j++ ) {
            /* block 3 */
            if (RationalGT(num[j], den[j],
                           num[j+1], den[j+1])
                  ) {
                /* block 4 */
                temp = num[j];
                num[j] = num[j+1];
                num[j+1] = temp; } } }
}
```

Figure 1: A faulty C function for sorting rational numbers

2.2 Program Spectra

A program spectrum [11] is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program. As such, recording a program spectrum is a light-weight analysis compared to other run-time methods, such as, e.g., dynamic slicing [10].

As an example, a *block count spectrum* tells how often each block of code is executed during a run of a program. In this paper, a block of code is a C language statement, where we do not distinguish between the individual statements of a compound statement, but where we do distinguish between the cases of a switch statement[1]. Suppose that the function `RationalSort` of Figure 1 is used to sort the sequence $\langle \frac{2}{1}, \frac{3}{1}, \frac{4}{1}, \frac{1}{1} \rangle$, which it happens to do correctly. This would result in the following block count spectrum, where block 5 refers to the body of the `RationalGT` function, which has not been shown in Figure 1.

block	1	2	3	4	5
count	1	4	6	3	6

Block 1, the body of the function `RationalSort`, is executed once. Blocks 2 and 3, the bodies of the two loops, are executed four and six times, respectively. To sort our example array, three exchanges must be made, and block 4, the body of the conditional statement, is executed three times. Block 5, the `RationalGT` function body, is executed six times: once for every iteration of the inner loop.

If we are only interested in whether a block is executed or not, we can use binary flags instead

[1] This is a slightly different notion than a *basic block*, which is a block of code that has no branch.

$$\begin{array}{ccc} & N \text{ parts} & \text{errors} \\ M \text{ spectra} & \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \\ & \begin{matrix} s_1 & s_2 & \dots & s_N \end{matrix} & \end{array}$$

Figure 2: The ingredients of fault diagnosis

of counters. In this case, the block count spectra revert to block *hit* spectra. Beside block count/hit spectra, many other forms of program spectra exist. See [7] for an overview. In this paper we will work with block hit spectra, and hit spectra for logical threads used in the software of our test case (see Section 4.1).

2.3 Fault Diagnosis

The hit spectra of M runs constitute a binary matrix, whose columns correspond to N different parts of the program (see Figure 2). In our case, these parts are blocks of C code. In some of the runs an error is detected. This information constitutes another column vector, the error vector. This vector can be thought of as to represent a hypothetical part of the program that is responsible for all observed errors. Fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program spectra, are quantified by means of *similarity coefficients* (see, e.g., [8]). As an example, the Jaccard similarity coefficient (see also [8]) expresses the similarity s_j of column j and the error vector as the number of positions in which these vectors share an entry 1 (i.e., block was exercised and the run has failed), divided by this same number plus the number of positions in which the vectors have different entries:

$$s_j = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)} \tag{1}$$

where $a_{pq}(j) = |\{i \mid x_{ij} = p \wedge e_i = q\}|$, and $p, q \in \{0, 1\}$.

Under the assumption that a high similarity to the error vector indicates a high probability that the corresponding parts of the software cause the detected errors, the calculated similarity coefficients rank the parts of the program with respect to their likelihood of containing the faults.

To illustrate the approach, suppose that we apply the `RationalSort` function to the input sequences $I_1 = \langle\ \rangle$, $I_2 = \langle \frac{1}{4} \rangle$, $I_3 = \langle \frac{2}{1}, \frac{1}{1} \rangle$ and $I_4 = \langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, $I_5 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$, and $I_6 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$.

I_1, I_2, and I_6 are already sorted, and lead to passed runs. I_3 is not sorted, but the denominators in this sequence happen to be equal, in which case no error occurs. I_4 is the example from Section 2.1: it is not sorted, and an error occurs during its execution, but this error goes undetected. Only for I_5 the program fails. The calculated result is $\langle \frac{1}{1}, \frac{2}{2}, \frac{4}{3}, \frac{3}{4} \rangle$ instead of $\langle \frac{1}{4}, \frac{2}{2}, \frac{4}{3}, \frac{3}{1} \rangle$, which is a clear indication that an error has occurred.

The block hit spectra for these runs are as follows ('1' denotes a hit), where block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Figure 1.

	block					
input	1	2	3	4	5	error
I_1	1	0	0	0	0	0
I_2	1	1	0	0	0	0
I_3	1	1	1	1	1	0
I_4	1	1	1	1	1	0
I_5	1	1	1	1	1	1
I_6	1	1	1	0	1	0

For this data, the calculated Jaccard coefficients are $s_1 = \frac{1}{6}$, $s_2 = \frac{1}{5}$, $s_3 = \frac{1}{4}$, $s_4 = \frac{1}{3}$, $s_5 = \frac{1}{4}$, which (correctly) identifies block 4 as the most likely location of the fault.

3 Relevance to Embedded Software

The effectiveness of the diagnosis technique described in the previous section has already been demonstrated in several articles (see, e.g., [1], [4], [9]). In this paper we present the benefits and discuss the issues specifically related to debugging embedded software in consumer electronics products. Especially because of constraints imposed by the market, the conditions under which this software is developed are somewhat different from those for other software products:

- To reduce unit costs, and often to ensure portability of the devices, the software runs on non-commodity hardware, and computing resources are limited.
- As a consequence, many facilities that developers of non-embedded software have come to rely on are absent, or are available only in rudimentary forms. Examples are profiling tools that give insight in the dynamic behavior of systems.
- At the same time, the systems are highly concurrent, and operate at a low level of abstraction from the hardware. Therefore, their design and implementation are complicated by

factors that can largely be abstracted away from in other software systems, such as deadlock prevention, and timing constraints involved in, e.g., writing to the graphics display only in those fractions of a second that the screen is not being refreshed.

- On top of challenges that the entire software industry has to deal with, such as geographically distributed development organizations, the strong competition between manufacturers of consumer electronics makes it absolutely vital that release deadlines are met.

- Although important safety mechanisms, such as short-circuit detection, are sometimes implemented in software, for a large part of the functionality there are no personal risks involved in transient failures.

Consequently, it is not uncommon that consumer electronics products are shipped with several known software faults outstanding. To a certain extent, this also holds for other software products, but the combination of the complexity of the systems, the tight constraints imposed by the market, and the relatively low impact of the majority of possible system failures creates a unique situation. Instead of aiming for correctness, the goal is to create a product that is of value to customers, despite its imperfections, and to bring the reliability to a commercially acceptable level (also compared to the competition) before a product must be released.

The technique of Section 2 can help to reach this goal faster, and may thus reduce the time-to-market, and lead to more reliable products. Specific benefits are the following.

- As a black-box diagnosis technique, it can be applied without any additional modeling effort. This effort would be hard to justify under the market conditions described above. Moreover, concurrent systems are difficult to model.

- The technique improves insight in the run-time behavior. For embedded software in consumer electronics, this is often lacking, because of the concurrency, but also because of the decentralized development.

- We expect that the technique can easily be integrated with existing testing procedures, such as overnight playback of recorded usage scenarios. In addition to the information that errors have occurred in some scenarios, this gives a first indication of the parts of the software that are likely to be involved in these errors. In the large, geographically distributed development organizations that we are dealing with, it may also help to identify which teams of developers to contact.

- Last but not least, the technique is lightweight, which is relevant because of the non-commodity hardware and limited computing resources. All that is needed is some memory for storing program spectra, or for calculating the similarity coefficients on the fly (which reduces the space complexity from $O(M \times N)$ to $O(N)$, see Section 5). Profiling tools such as `gcov` are convenient for obtaining program spectra, but they are typically not available in a development environment for embedded software. However, the same data can be obtained through source code instrumentation.

While none of these benefits are unique, their combination makes program spectrum analysis an attractive technique for diagnosing embedded software in consumer electronics.

4 Experiments

In this section we describe our experience with applying the techniques of Section 2 to an industrial test case.

4.1 Platform

The subject of our experiments is the control software in a particular product line of analog television sets. All audio and video processing is implemented in hardware, but the software is responsible for tasks such as decoding remote control input, displaying the on-screen menu, and coordinating the hardware (e.g., optimizing parameters for audio and video processing based on an analysis of the signals). Most teletext[2] functionality is also implemented in software.

The software itself consists of approximately 450K lines of C code, which is configured from a much larger (several MLOC) code base of Koala software components [12].

The control processor is a MIPS running a small multi-tasking operating system. Essentially, the run-time environment consists of several threads with increasing priorities, and for synchronization purposes, the work on these threads is organized in 315 logical threads inside the various components. Threads are preempted when work arrives for a higher-priority thread.

[2] A standard for broadcasting information (e.g., news, weather, TV guide) in text pages, very popular in Europe.

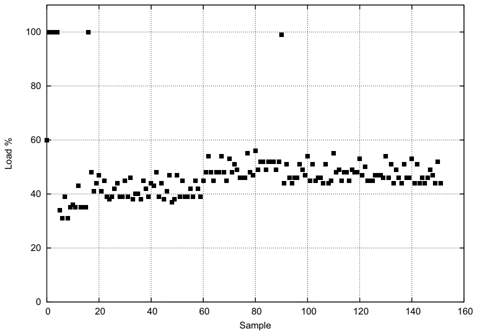


Figure 3: CPU load measured per second

The total available RAM memory in consumer sets is two megabyte, but in the special developer version that we used for our experiments, another two megabyte was available. In addition, the developer sets have a serial connection, and a debugger interface for manual debugging on a PC.

4.2 Faults

We diagnosed two faults, one existing, and one that was seeded to reproduce an error from a different product line.

Load Problem. A known problem with the specific version of the control software that we had access to, is that after teletext viewing, the CPU load when watching television (TV mode) is approximately 10% higher than before teletext viewing. This is illustrated in Figure 3, which shows the CPU load for the following scenario: one minute TV mode, 30 s teletext viewing, and one minute of TV mode. The CPU load clearly increases around the 60th sample, when the teletext viewing starts, but never returns to its initial level after sample 90, when we switch back to TV mode.

Teletext Lock-up Problem. Another product line of television sets provides a function for searching in teletext pages. An existing fault in this functionality entails that searching in a page without visible content locks up the teletext system. A likely cause for the lock-up is an inconsistency in the values of two state variables in different components, for which only specific combinations are allowed. We hard-coded a remote control key-sequence that injects this error on our test platform.

4.3 Implementation

We wrote a small Koala component for recording and storing program spectra, and for transmitting them off the television set via the serial connection. The transmission is done on a low-priority thread while the CPU is otherwise idle, in order to minimize the impact on the timing behavior. Pending their transmission via the serial connection, our component caches program spectra in the extra memory available in our developer version of the hardware.

For diagnosing the load problem we obtained hit spectra for the logical threads mentioned in Section 4.1, resulting in spectra of 315 binary flags. We approached the lock-up problem at a much finer granularity, and obtained block hit spectra for practically all blocks of code in the control software, resulting in spectra of over 60,000 flags.

The hit spectra for the logical threads are obtained by manually instrumenting a centralized scheduling mechanism. For the block hit spectra we automatically instrumented the entire source code using the Front [2] parser generator.

In Section 2.3 we use program spectra for different runs of the software, but for embedded software in consumer electronics, and indeed for most interactive systems, the concept of a run is not very useful. Therefore we record the spectra per *transaction*, instead of per run, and we use two different notions of a transaction for the two different faults that we diagnosed:

- for the load problem, we use a periodic notion of a transaction, and record the spectra per second.
- for the lock-up problem, we define a transaction as the computation in between two key-presses on the remote control.

4.4 Diagnosis

For the load problem we used the scenario of Figure 3. We marked the last 60 spectra, for the second period of TV mode as 'failed,' and those of earlier transactions as 'passed.' In the ranking that follows from the analysis of Section 2.3, the logical thread that had been identified by the developers as the actual cause of the load problem was in the second position out of 315. In the first position was a logical thread related to teletext, whose activation is part of the problem, so in this case we can conclude that although the diagnosis is not perfect, the implied suggestion for investigating the problem is quite useful.

For the lock-up problem, we used a proper error detection mechanism. On each key-press, when caching the current spectrum, a separate routine verifies the values of the two state variables, and marks the current spectrum as failed if they assume an invalid combination. Although this is a special-purpose mechanism, including and regularly checking high-level assert-like statements about correct behavior is a valid means to increase the error-awareness of systems.

Using a very simple scenario of 23 key-presses that essentially (1) verifies that the TV and teletext subsystems function correctly, (2) triggers the error injection, and (3) checks that the teletext subsystem is no longer responding, we immediately got a good diagnosis of the detected error: the first two positions in the total ranking of over 60,000 blocks pointed directly to our error injection code. Adding another three key-presses to exonerate an uncovered branch in this code made the diagnosis perfect: the exact statement that introduced the state inconsistency was located out of approximately 450K lines of source code.

5 Discussion

Especially the results for the lock-up problem have convinced us that program spectra, and their application to fault diagnosis are a viable technique and useful tool in the area of embedded software in consumer electronics. However, there are a number of issues with our implementation.

First, we cannot claim that we have not altered the timing behavior of the system. Because of its rigorous design, the TV is still functioning properly, but everything runs much slower with the block-level instrumentation (e.g., changing channels now takes seconds). One reason is that currently, we collect block *count* spectra at byte resolution, and convert to block *hit* spectra off-line. Updating the counters in a multi-threaded environment requires a critical section for every executed block, which is hugely expensive. Fortunately, this information is not needed, and we believe we can implement a binary flag update without a critical section.

Second, we cache the spectra of passed transactions, and transmit them off the system during CPU idle time. Because of the low throughput of the serial connection, this may become a bottleneck for large spectra and larger scenarios. In our case we could store 25 spectra of 65,536 counters, which was already slowing down the scenarios with more than that number of transactions, but even with a more memory-efficient implementation, this inevitably becomes a problem with, for example, overnight testing.

For many purposes, however, we will not have to store the actual spectra. In particular for fault diagnosis, ultimately we are only interested in the calculated similarity coefficients, and all similarity coefficients that we are aware of are expressed in terms of the four counters a_{00}, a_{01}, a_{10}, and a_{11} introduced in Section 2.3. If an error detection mechanism is available, like in our experiments with the lock-up problem, then these four counters can be calculated on the fly, and the memory requirements become linear in the number columns in the matrix of Figure 2.

6 Related Work

Program spectra themselves were introduced in [11], where hit spectra of intra-procedural paths are analyzed to diagnose year 2000 problems. The distinction between count spectra and hit spectra is introduced in [7], where several kinds of program spectra are evaluated in the context of regression testing. In the introduction we already mentioned three practical diagnosis/debugging tools [4, 5, 9] that are essentially based on the same diagnosis method as ours. A recent study, reported in [1], indicates that the choice of the similarity coefficient, as introduced in Section 2.3 can be of significant influence on the quality of the diagnosis.

As we mentioned in Section 3, we are dealing with a black box diagnosis technique that can be applied without additional knowledge about a system. An example of a white box technique is model-based diagnosis (see, e.g., [6]), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. White box approaches to software diagnosis exist (see, e.g., [13]), but software modeling is extremely complex, so most software diagnosis techniques are black box.

7 Conclusion

In this paper we have demonstrated software fault diagnosis through the analysis of program spectra, on a large-scale industrial test case in the area of embedded software in consumer electronics devices. In addition to confirming established effectiveness results, our experiments indicate that the technique lends itself well for application in the resource-constrained environments that are typical for the development of embedded software.

While our current experiments focus on development-time debugging, they open corridors to further applications, such as run-time recovery by rebooting only those parts of a system whose activities correlate with detected errors.

8 Acknowledgments

We would like to thank Pierre van de Laar for valuable comments on an earlier version of this paper.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of PRDC'06*, pages 39–46. IEEE Computer Society, 2006.

[2] L. Augusteijn. Front: a front-end generator for Lex, Yacc and C, release 1.0, 2002. See `http://front.sourceforge.net/`.

[3] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. of the 2002 Int. Conf. on Dependable Systems and Networks (DSN)*, pages 595–604. IEEE Computer Society, 2002.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In A. P. Black, editor, *Proceedings of ECOOP 2005*, volume 3586 of *LNCS*, pages 528–550. Springer-Verlag, 2005.

[6] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[7] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *ACM SIGPLAN Notices*, 33(7):83–90, 1998.

[8] A. K. Jain and R. C. Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., 1988.

[9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of ICSE 2002*, pages 467–477. ACM Press, 2002.

[10] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[11] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of ESEC/FSE 97*, volume 1301 of *LNCS*, pages 432–449. Springer–Verlag, 1997.

[12] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, March 2000.

[13] F. Wotawa, M. Strumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In T. Hendtlass and M. Ali, editors, *Proceedings of IAE/AIE 2002*, volume 2358 of *LNCS*, pages 746–757. Springer-Verlag, 2002.

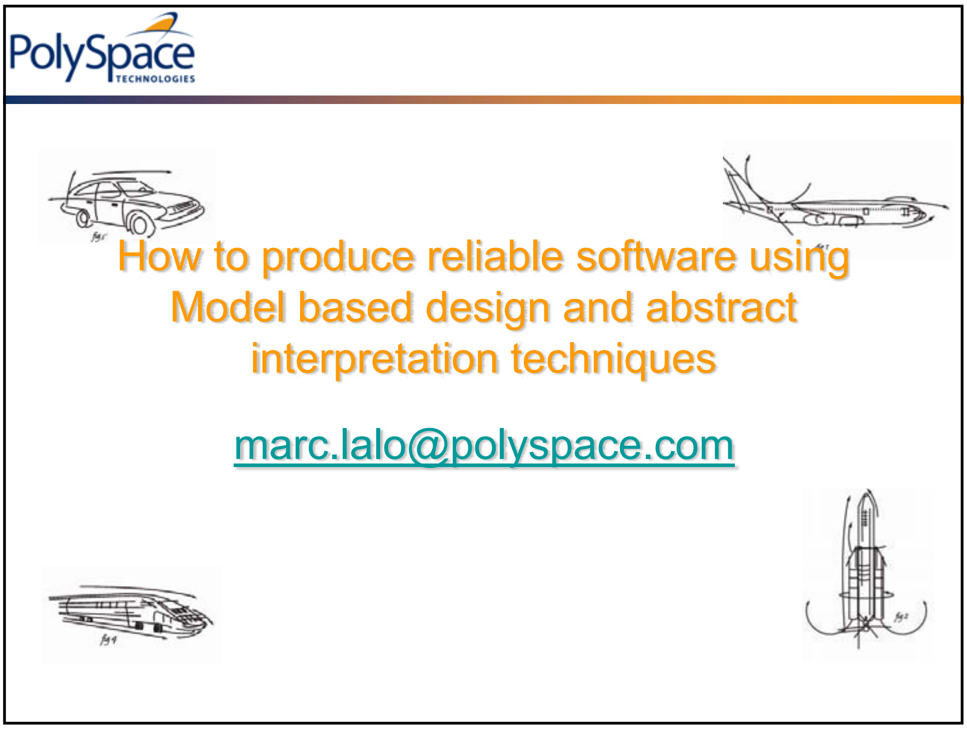
About the Authors

Peter Zoeteweij works in the Software Engineering Research Group at Delft University of Technology. He holds an MSc. from Delft University of Technology, and a PhD. from the University of Amsterdam, both in computer science. Before his PhD., Peter worked for several years as a software engineer for Logica (now LogicaCMG), mainly on software for the oil industry.

Rui Abreu is a PhD. student at the Embedded Software Lab within the Software Engineering Research Group at Delft University of Technology. He holds an MSc. in Computer Science and Systems Engineering from Minho University, Portugal. Through his thesis work at Siemens R&D Porto, and professional internship at Philips Research, he acquired industrial experience in the area of embedded systems.

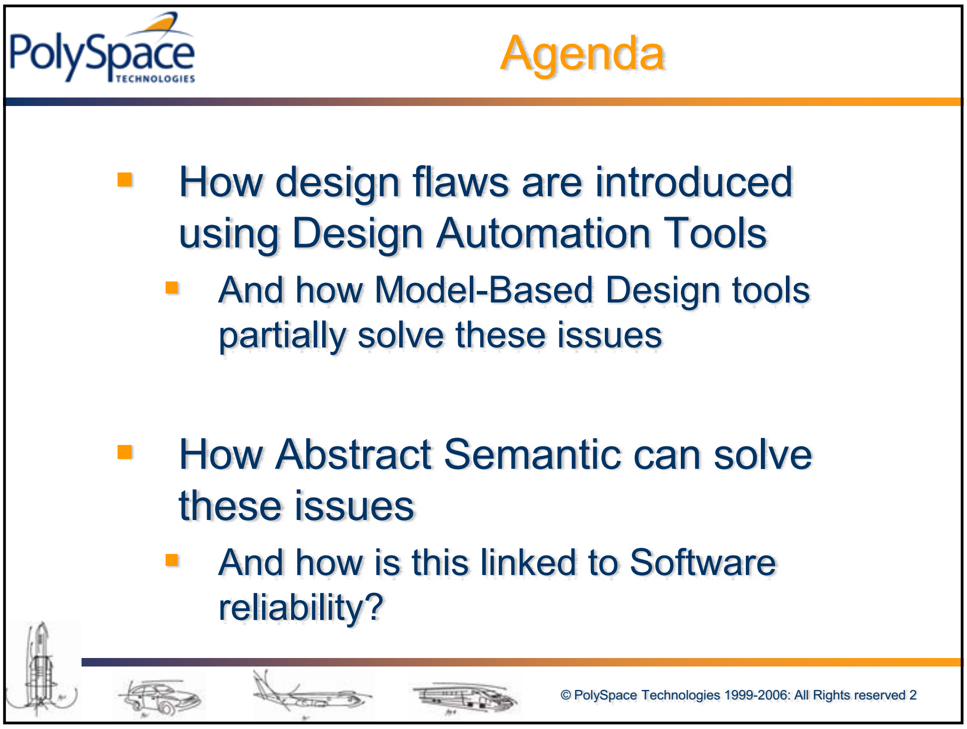
Rob Golsteijn holds an MSc. in Computing Science from Eindhoven University of Technology and completed the two years' post-graduate Software Technology program from the Stan Ackermans Institute. Rob now works for NXP, formerly known as Philips Semiconductors, and has experience in embedded software development of television platforms and products. Rob is currently working as a member of an industrial research project focusing on reliability of resource-constrained consumer devices.

Arjan J.C. van Gemund holds a BSc. in physics, and an MSc. (cum laude) and PhD. (cum laude) in computer science, all from Delft University of Technology. He has held positions at DSM and TNO, and currently serves as a full professor at the Electrical Engineering, Mathematics, and Computer Science Faculty of Delft University of Technology, heading the Embedded Software Lab within the Software Engineering Research Group.



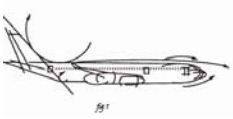
How to produce reliable software using Model based design and abstract interpretation techniques

marc.lalo@polyspace.com



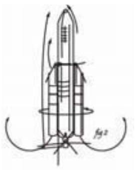
Agenda

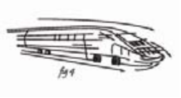
- How design flaws are introduced using Design Automation Tools
  - And how Model-Based Design tools partially solve these issues
- How Abstract Semantic can solve these issues
  - And how is this linked to Software reliability?



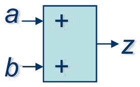
How design flaws are introduced

And how Model-Based Design
tools partially solve these issues

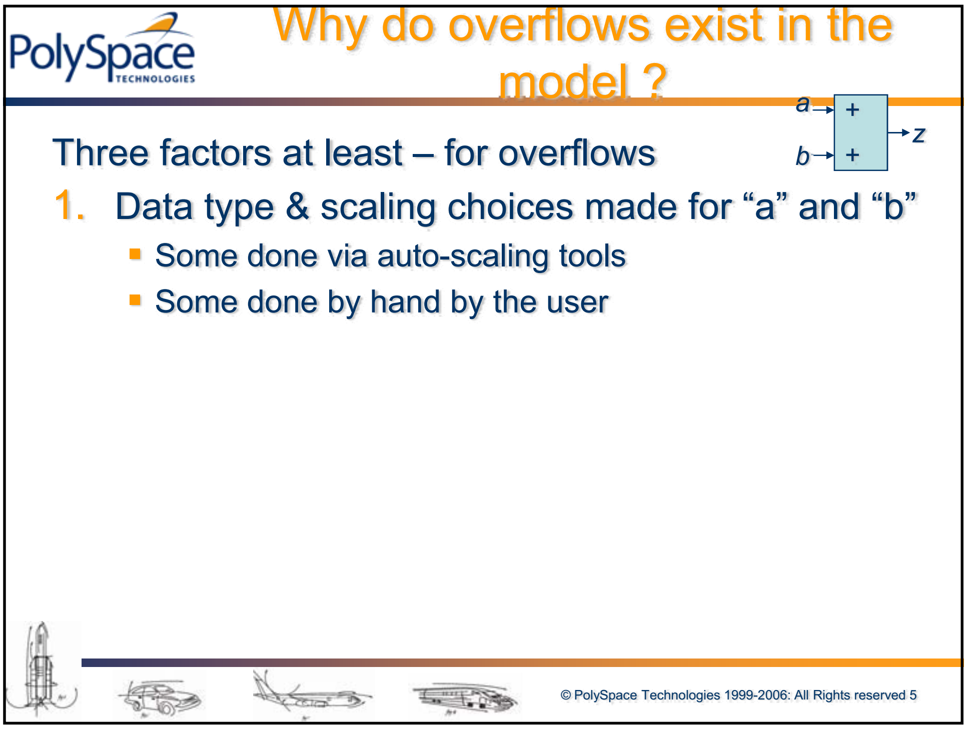




Starting from an example



- Consider a simple example
  - The user performs a “z = a + b” operation
  - “a” and “b” may be entries of the model
  - “a” and “b” ranges may be known and specified at the model level
- Automatic code generator generates
  z = a + b;
- The code generator could saturate the output
  - Leads to an inefficient code (size, speed) given all operations potentially need saturation

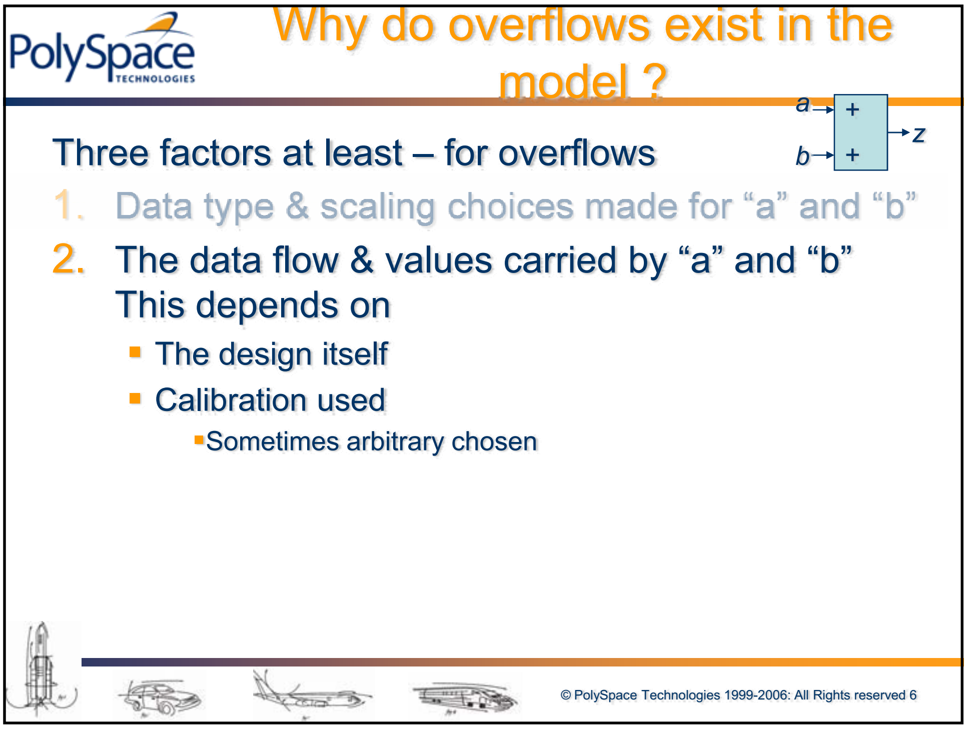


Why do overflows exist in the model ?

Three factors at least – for overflows

1. Data type & scaling choices made for “a” and “b”
   - Some done via auto-scaling tools
   - Some done by hand by the user

© PolySpace Technologies 1999-2006: All Rights reserved 5

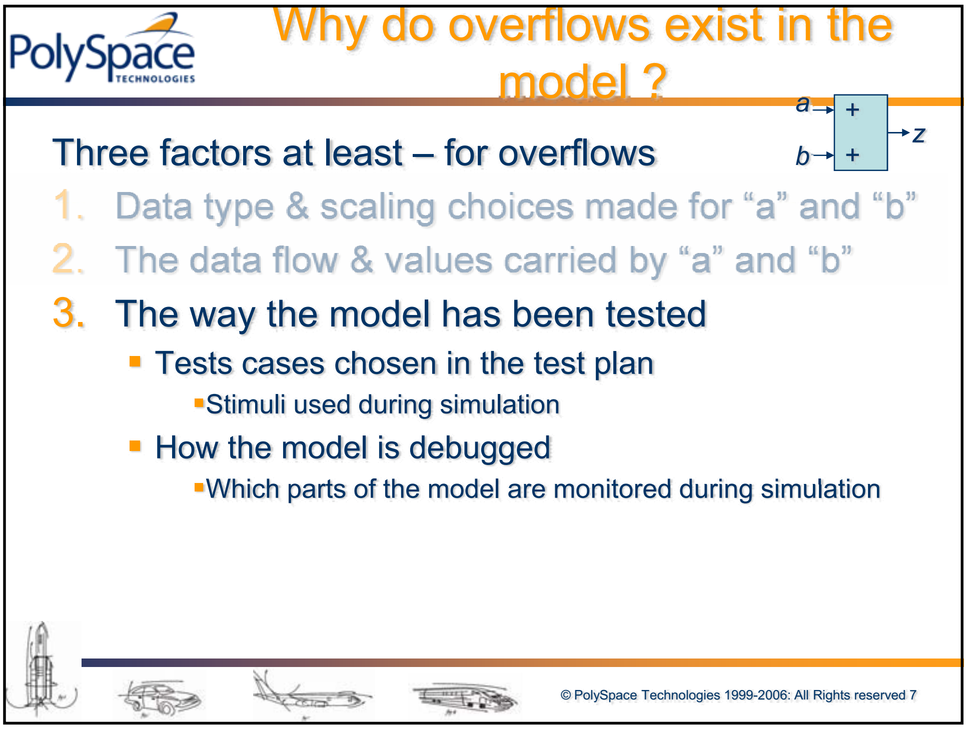


Why do overflows exist in the model ?

Three factors at least – for overflows

1. Data type & scaling choices made for “a” and “b”
2. The data flow & values carried by “a” and “b”
   This depends on
   - The design itself
   - Calibration used
     - Sometimes arbitrary chosen

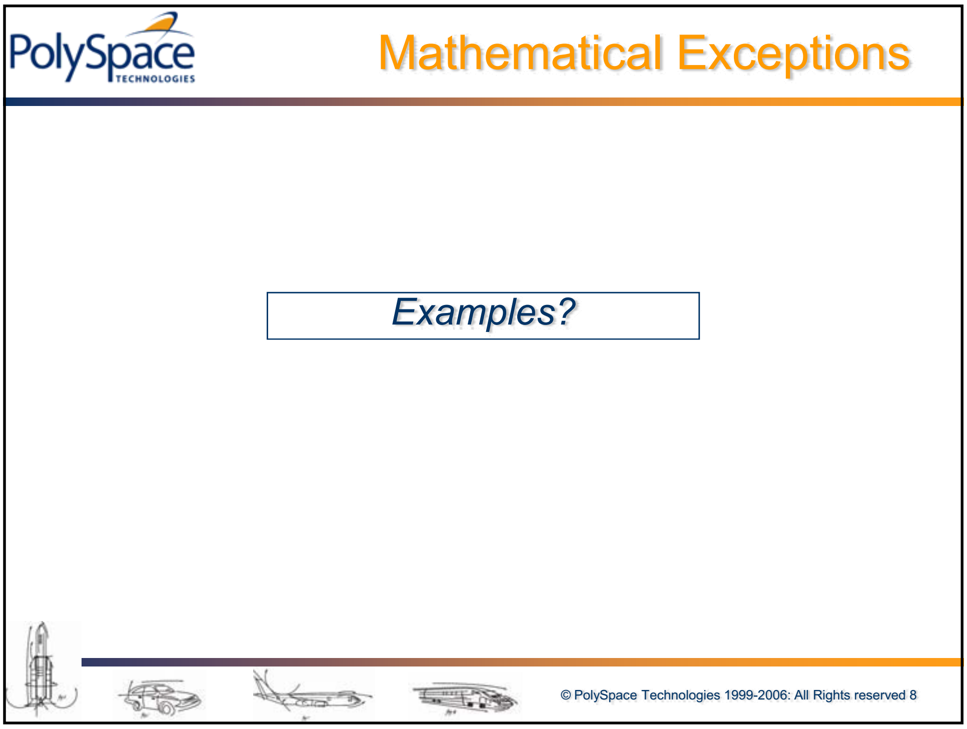
© PolySpace Technologies 1999-2006: All Rights reserved 6



Why do overflows exist in the model ?

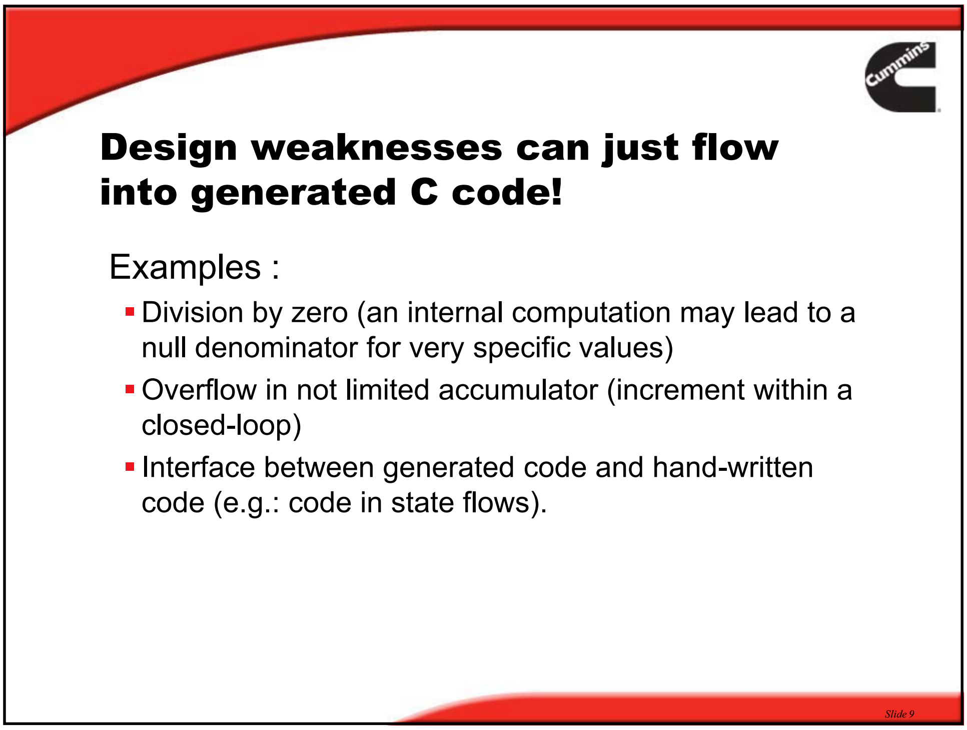
Three factors at least – for overflows

1. Data type & scaling choices made for “a” and “b”
2. The data flow & values carried by “a” and “b”
3. The way the model has been tested
   - Tests cases chosen in the test plan
     - Stimuli used during simulation
   - How the model is debugged
     - Which parts of the model are monitored during simulation



Mathematical Exceptions

Examples?

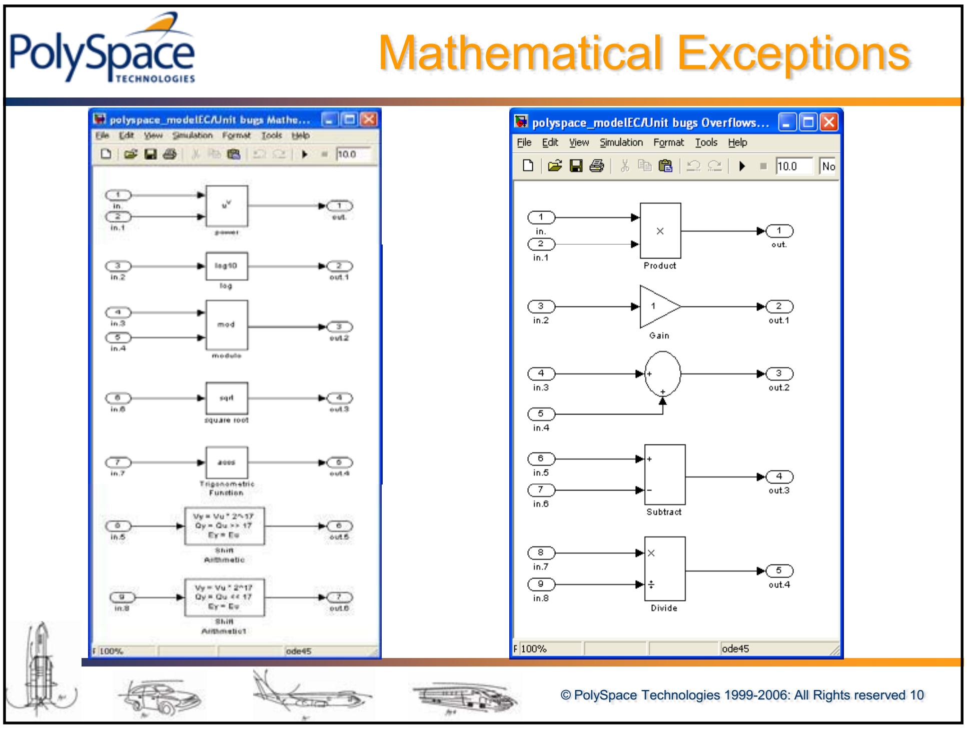


Design weaknesses can just flow into generated C code!

Examples :

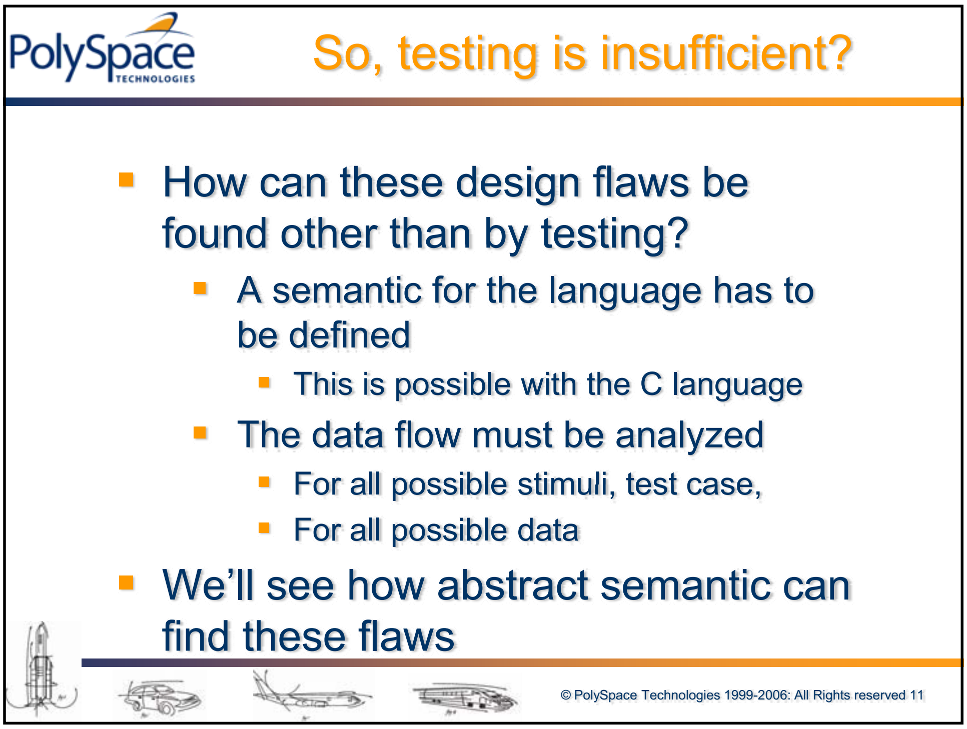
- Division by zero (an internal computation may lead to a null denominator for very specific values)
- Overflow in not limited accumulator (increment within a closed-loop)
- Interface between generated code and hand-written code (e.g.: code in state flows).

Slide 9



Mathematical Exceptions

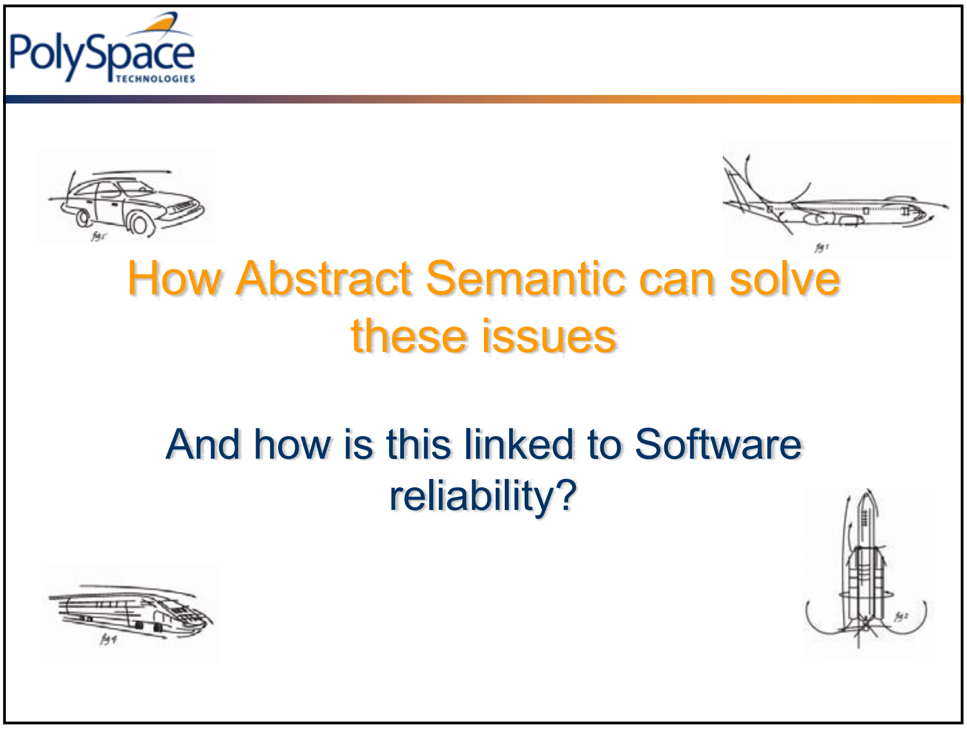
© PolySpace Technologies 1999-2006: All Rights reserved 10



PolySpace TECHNOLOGIES

So, testing is insufficient?

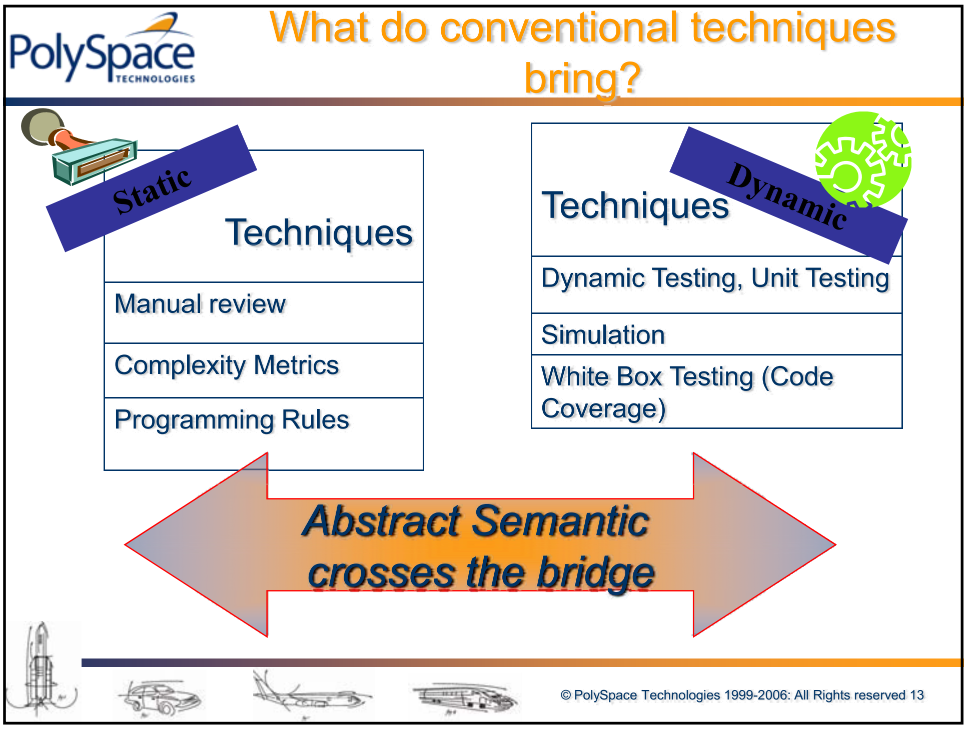
- How can these design flaws be found other than by testing?
  - A semantic for the language has to be defined
    - This is possible with the C language
  - The data flow must be analyzed
    - For all possible stimuli, test case,
    - For all possible data
- We'll see how abstract semantic can find these flaws



PolySpace TECHNOLOGIES

How Abstract Semantic can solve these issues

And how is this linked to Software reliability?



What do conventional techniques bring?

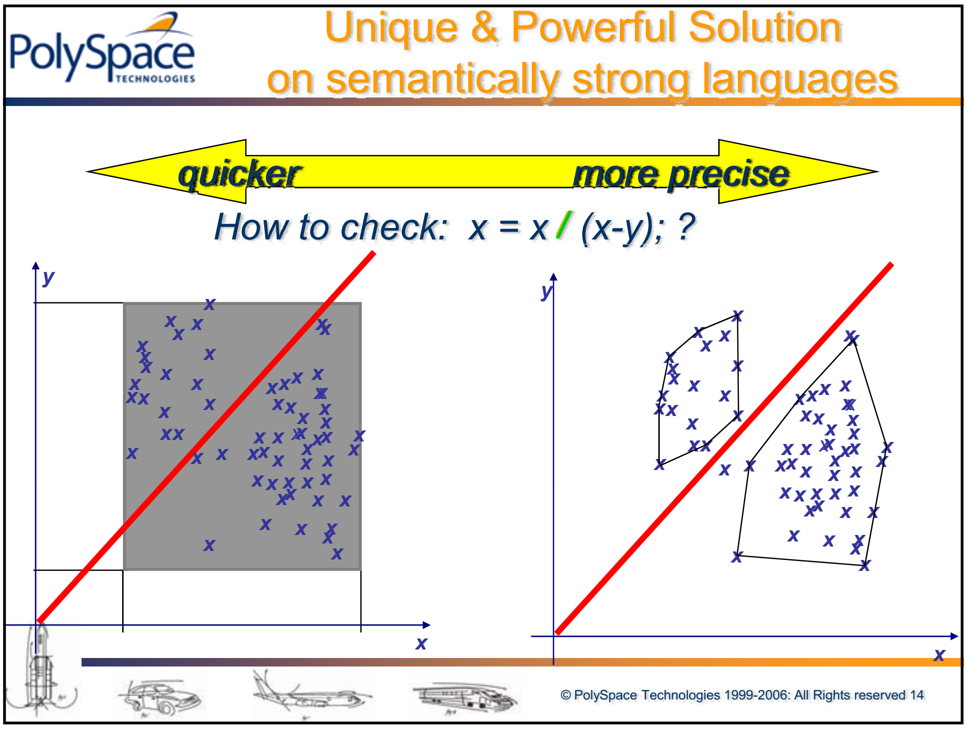
Static Techniques

- Manual review
- Complexity Metrics
- Programming Rules

Dynamic Techniques

- Dynamic Testing, Unit Testing
- Simulation
- White Box Testing (Code Coverage)

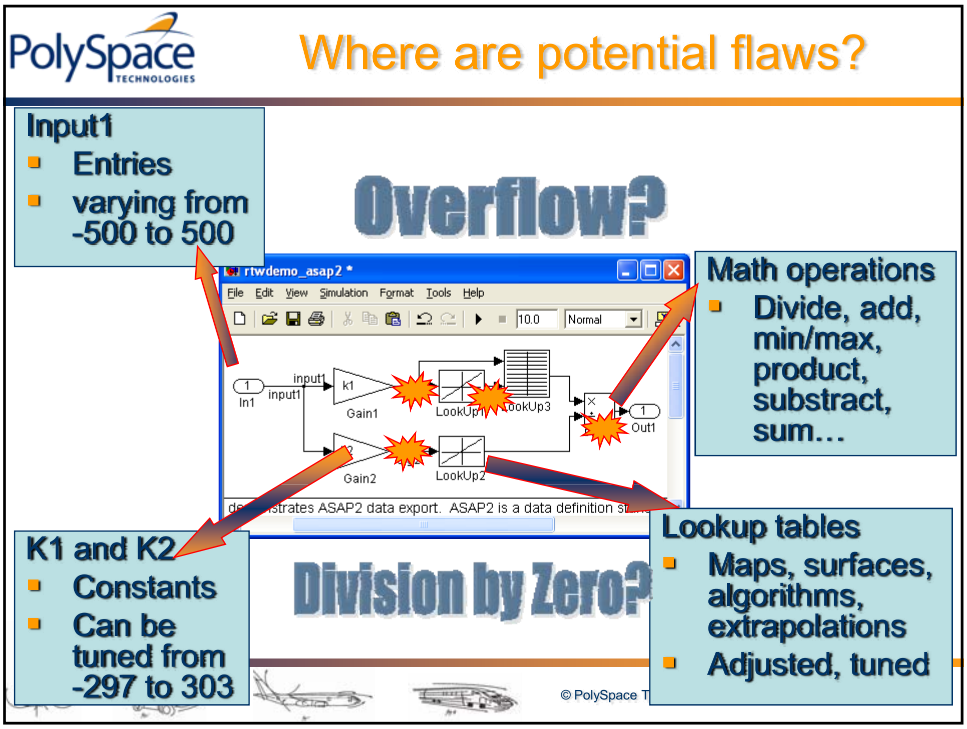
Abstract Semantic crosses the bridge



Unique & Powerful Solution on semantically strong languages

quicker ⟷ *more precise*

How to check: x = x / (x-y); ?



Where are potential flaws?

Input1
- Entries
- varying from -500 to 500

Overflow?

Math operations
- Divide, add, min/max, product, substract, sum…

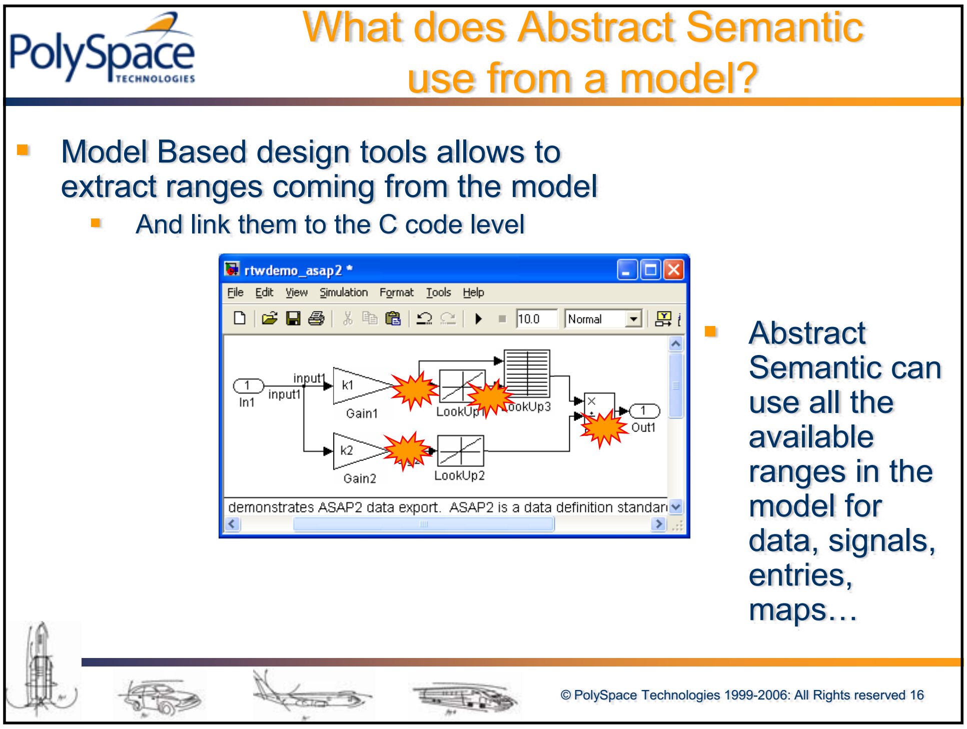
K1 and K2
- Constants
- Can be tuned from -297 to 303

Division by Zero?

Lookup tables
- Maps, surfaces, algorithms, extrapolations
- Adjusted, tuned

de strates ASAP2 data export. ASAP2 is a data definition st

© PolySpace T



What does Abstract Semantic use from a model?

- Model Based design tools allows to extract ranges coming from the model
  - And link them to the C code level

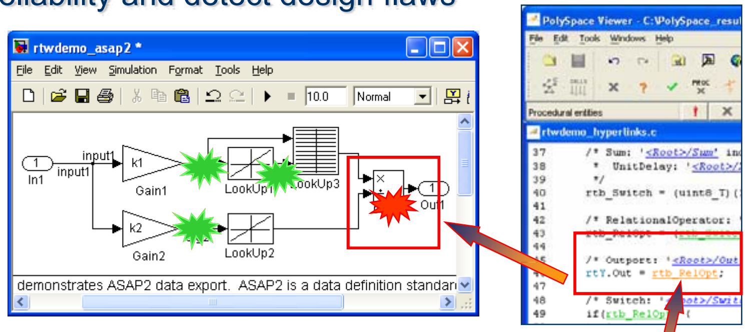
- Abstract Semantic can use all the available ranges in the model for data, signals, entries, maps…

demonstrates ASAP2 data export. ASAP2 is a data definition standar

© PolySpace Technologies 1999-2006: All Rights reserved 16

Abstract Semantic results

- Abstract Semantic can work on auto-generated code – C or C++ language – to prove reliability and detect design flaws



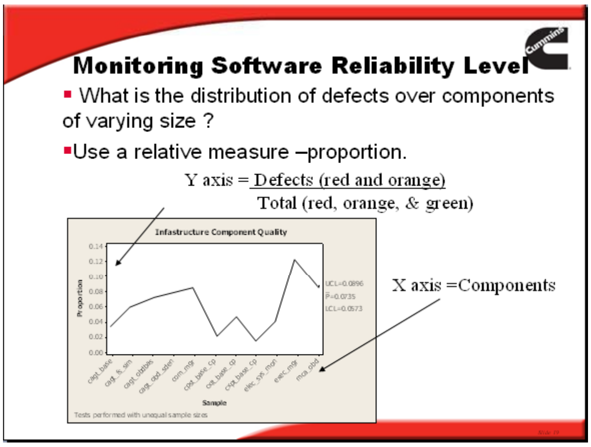
demonstrates ASAP2 data export. ASAP2 is a data definition standar

```
/* Sum: '<Root>/Sum' in
 *  UnitDelay: '<Root>/
 */
rtb_Switch = (uint8_T)(

/* RelationalOperator: '
rtb_RelOpt = (rtb_Switc

/* Outport: '<Root>/Out
rtY.Out = rtb_RelOpt;

/* Switch: '<Root>/Swit
if(rtb_RelOpt {
```

Abstract Semantic can detect an error here (after having analyzed of the generated code)

And measures?

- How does it impact reliabiltiy?

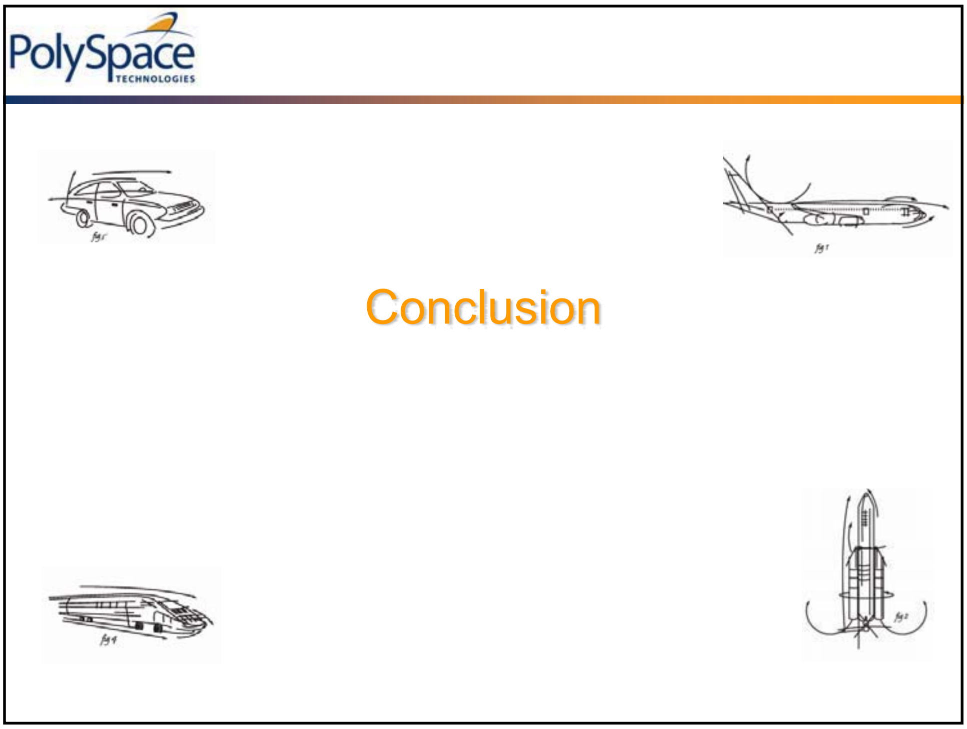


Monitoring Software Reliability Level

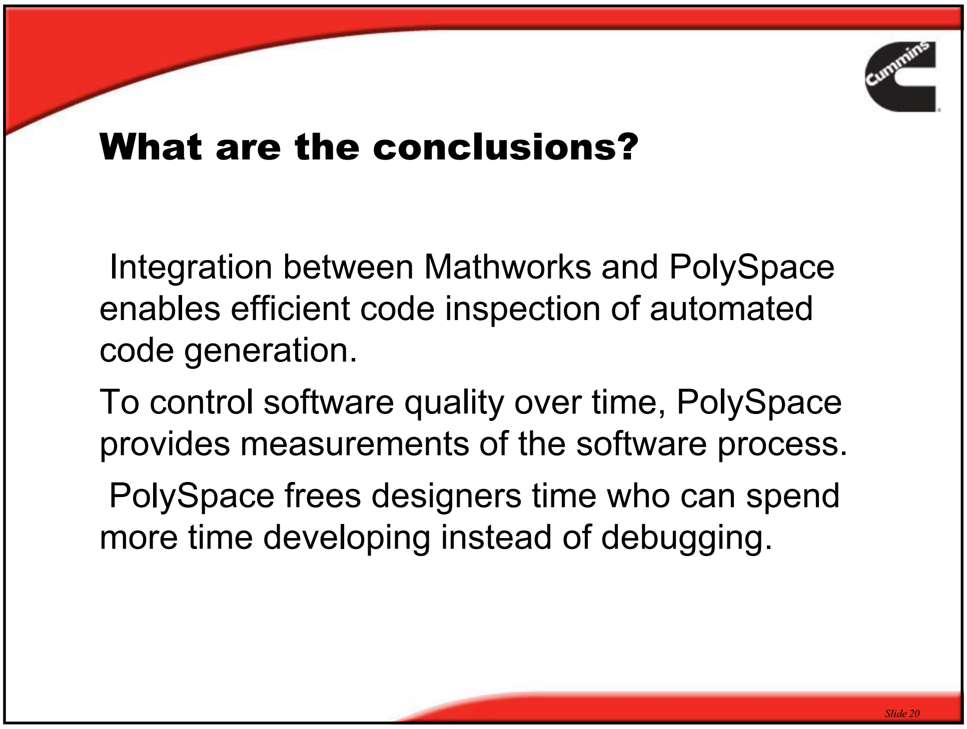
- What is the distribution of defects over components of varying size ?
- Use a relative measure –proportion.

Y axis = Defects (red and orange) / Total (red, orange, & green)

X axis =Components



Conclusion



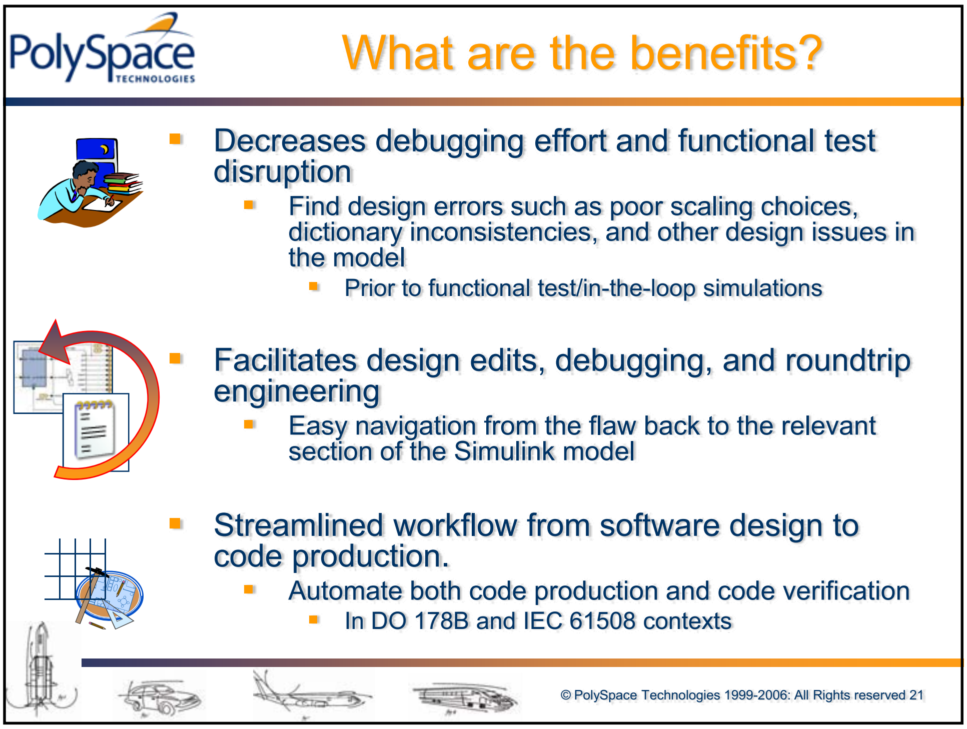
What are the conclusions?

Integration between Mathworks and PolySpace enables efficient code inspection of automated code generation.

To control software quality over time, PolySpace provides measurements of the software process.

PolySpace frees designers time who can spend more time developing instead of debugging.

Slide 20



What are the benefits?

- Decreases debugging effort and functional test disruption
  - Find design errors such as poor scaling choices, dictionary inconsistencies, and other design issues in the model
    - Prior to functional test/in-the-loop simulations
- Facilitates design edits, debugging, and roundtrip engineering
  - Easy navigation from the flaw back to the relevant section of the Simulink model
- Streamlined workflow from software design to code production.
  - Automate both code production and code verification
    - In DO 178B and IEC 61508 contexts

© PolySpace Technologies 1999-2006: All Rights reserved 21

And you?
How would you solve these challenges?

Thanks for your attention!!!

www.polyspace.com

Automated Software Testing and Release with Nix Build Farms

Eelco Dolstra
Utrecht University
eelco@cs.uu.nl

Eelco Visser
Delft University of Technology
visser@acm.org

1 Introduction

Continuous integration [7] is a good software engineering practice. The idea is that each software development project should have a fully automated build system. We can then run the build system automatically to continuously produce the most recent version of the software. Every time a developer commits a change to the project's version management system, the continuous integration system checks out the source of the project, runs its automated build process, and creates a report describing the result of the build. The latter might be presented on a web page and/or sent to the developers through e-mail.

Of course, developers are supposed to test their changes before they commit. The added advantage of a continuous integration system (apart from catching developers who *do not* test their changes) is that it allows much more in-depth testing of the component(s) being developed:

- The software may need to be built and tested on many different platforms (i.e., *portability testing*). It is infeasible for each developer to do this before every commit.
- Likewise, many projects have very large test sets (e.g., regression tests in a compiler, or stress tests in a DBMS) that can take hours or days to run to completion.
- Many kinds of static and dynamic analyses can be performed as part of the tests, such as code coverage runs and static analyses.
- It may also be necessary to build many different *variants* of the software. For instance, it may be necessary to verify that the component builds with various versions of a compiler.
- Developers typically use incremental building to test their changes (since a full build may take too long), but this is unreliable with many build management tools (such as Make), i.e., the result of the incremental build might differ from a full build.

- It ensures that the software can be built from the sources under revision control. Users of version management systems such as CVS and Subversion often forget to place source files under revision control.

- The machines on which the continuous integration system runs ideally provides a clean, well-defined build environment. If this environment is administered through proper SCM techniques, then builds produced by the system can be reproduced. In contrast, developer work environments are typically not under any kind of SCM control.

- In large projects, developers often work on a particular component of the project, and do not build and test the composition of those components (again since this is likely to take too long). To prevent the phenomenon of "big bang integration", where components are only tested together near the end of the development process, it is important to test components together as soon as possible (hence *continuous integration*).

A continuous integration system can also produce *releases* automatically. That is, when an automatic build succeeds (and possibly when it fails!), the build result can be packaged and made available in some way to developers and users. For instance, it can produce a web page containing links to the packaged source code for the release, as well as binary distributions. The production of releases fits naturally in the actions described above: the build process of the component should produce the desired release artifacts, and the presentation of the build result will be the release web page.

2 Build farms

The machines on which the continuous integration system runs are sometimes referred to as a *build farm* [8], since to support multi-platform projects or large sets of projects, a possibly large number of machines is required. In its simplest form, a build farm sits in a loop building and releasing software components from a version management system. These are its *jobs*. For each job, it performs the following tasks:

1. It obtains the latest version of the component's source code from the version management system.

2. It runs the component's build process (which presumably includes the execution of the component's test set).

3. It presents the results of the build (such as error logs) to the developers, e.g., by producing a web page.

Examples of build farms include CruiseControl [10], Tinderbox [6], Sisyphus [12] and Anthill [11]. However, these tools have various limitations:

- They do not manage the *build environment*. The build environment consists of the dependencies necessary to perform a build action, e.g., compilers, libraries, etc. Setting up the environment is typically done manually, and without proper SCM control (so it may be hard to reproduce a build at a later time). Manual management of the environment scales poorly in the number of configurations that must be supported.

  Suppose that we want to build a component that requires a certain compiler X. We then have to go to each machine and install X. If we later need a newer version of X, the process must be repeated all over again. An ever worse problem occurs if there are conflicting, mutually exclusive versions of the dependencies. Thus, simply installing the latest version is not an option. Of course, we can install these components in different directories and manually pass the appropriate paths to the build processes of the various components. But this is a rather tiresome and error-prone process.

- They do not easily support variability in software systems. A system may have a great deal of build-time variability: optional functionality, whether to build a debug or production version, different versions of dependencies, and so on. (For instance, the Linux kernel now has over 2,600 build-time configuration switches.) It is therefore important that a build farm can easily select and test different instances from the configuration space of the system to reveal problem, such as erroneous interactions between features. In a continuous integration setting, it is also useful to test different combinations of versions of subsystems, e.g., the head revision of a component against stable releases of its dependencies, and vice versa, as this can reveal various integration problems.

- A special case of variability is building many different *compositions* of versions of components, as this can reveal information useful from the perspective of continuous integration. Consider the real-life example shown in Figure 1 of two compilers: Stratego [13], a language based on strategic term rewriting; and Tiger, an implementation of the Tiger language [1] in Stratego. Suppose that both compilers have stable releases (e.g., Stratego 0.16 and Tiger 1.2) but are also under active development. Then there are various kinds of information that the Stratego and Tiger developers might want to obtain from the continuous integration system:

  - The Tiger developers want to know whether the most recent development version of Tiger (its *HEAD revision*) still builds. Here it is appropriate to build Tiger against a *stable* release of Stratego (i.e., 0.16), since when a build failure occurs the developers can then be reasonably certain that the cause is in Tiger, not Stratego.
  - However, the Tiger developers may also want to know whether their current source base is synchronised with possible changes in the Stratego compiler; i.e., whether the HEAD revision of Tiger builds against the HEAD revision of Stratego.

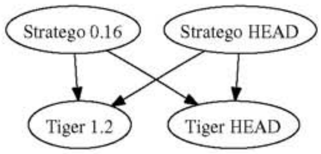


Figure 1: Multiple ways of combining component revisions

– Likewise, the Stratego developer may want to build the *stable* release of Tiger against the HEAD revision of Stratego, so that Tiger can act as a large regression test for Stratego.

This pattern is quite common in large projects where development is split among several development groups who every so often make releases available to other groups.

3 The Nix build farm

We have previously developed the *Nix deployment system* [5, 4, 3], which has precisely the properties needed to address the problems of managing the build environment and supporting variability. As a source-based deployment system, Nix has a *functional language* (the *Nix expression language*) to describe how to build components and how to compose them. This allows the build environment to be expressed in a self-contained and reproducible way, and it enables variability to be expressed by turning components into *functions* of the desired variabilities. The functional language also abstracts over multi-platform builds — Nix automatically dispatches the building of subexpressions to machines of the appropriate type.

Nix also stores components in such a way that variants of components do not interfere with each other (e.g., overwrite each other) and that prevents undeclared dependencies. The build result of each component instance is stored in the file system under a cryptographic hash of all inputs involved in building the component, such as its sources, build scripts, and dependencies such as compilers. For instance, a build of a particular job (Stratego/XT) might be stored under

```
/nix/store/09058krddyr8vwqk97yq...-strategoxt
```

If any input differs between two component build actions, then the resulting components will be stored in different locations in the file system and will not overwrite each other. Thus, conflicting dependencies such as different versions of a compiler no longer cause a problem; they are stored in isolation from each other. At the same time, if any two components between different build farm jobs have the same inputs, they will be built only once. This prevents unnecessary rebuilds.

```
<job id='patchelf-head'>
  <input id='job' type='svn'
    url='https://.../repos/trace/release/trunk' /> [1]
  <input id='patchelfHead' type='svn'
    url='https://.../repos/trace/patchelf/trunk />
  <job-script>generic-dist/build+upload.sh</job-script> [2]
  <arg>./jobs/nix/patchelf.nix</arg> [3]
  <arg>patchelfHeadRelease</arg> [4]
  <arg>http://nix.cs.uu.nl/dist/stratego</arg> [5]
  <notify-address>somebody@example.org</notify-address> [6]
</job>
```

Figure 2: A build job (simplified)

Contrary to most build farms (except [12]), the Nix build farm integrates testing and release management. That is, if a build succeeds, the result is a web page containing packages that can be installed directly by interested users through a variety of mechanisms, such as Linux RPM packages, Nix "channels", or Mac OS X installable packages.

The Nix build farm currently running at Utrecht University is used for a wide variety of open source packages, ranging from compiler suites such as the Stratego/XT program transformation system and the firmware of Philips televisions (in collaboration with Philips Research) to entire operating system distributions (the experimental NixOS). It runs on a variety of Unix platforms, Mac OS X, and Windows.

4 Implementation

This section gives a sketch of the build farm that we implemented using Nix. The build farm at present is not much more than a set of fairly simple scripts to run jobs, to build the desired release products such as various kinds of binary distributions, and to produce release pages. The "heavy lifting" of managing the environment is provided by Nix. Thus, the process of adding a job to the build farm consists essentially of writing Nix expressions that describe components and their dependencies.

The current Nix build farm consists of a number of components. At the highest level, there is a *supervisor* script (supervisor.pl) that reads build jobs from a file (jobs.conf) and executes them in circular order. The jobs file is in XML format. Figure 2 shows an example of the declaration of a build job for the HEAD revision of PatchELF, a small open source component developed by the first author. It specifies the locations of the inputs to the build (as URLs of Subversion repositories) $\boxed{1}$, the name of the script that performs the job $\boxed{2}$, and its command-line arguments $\boxed{3}$. Each input is fetched from its Subversion repository. The path of the job script is relative to the input that has ID job.

The supervisor sends e-mail notification if a job fails (or if it succeeds again

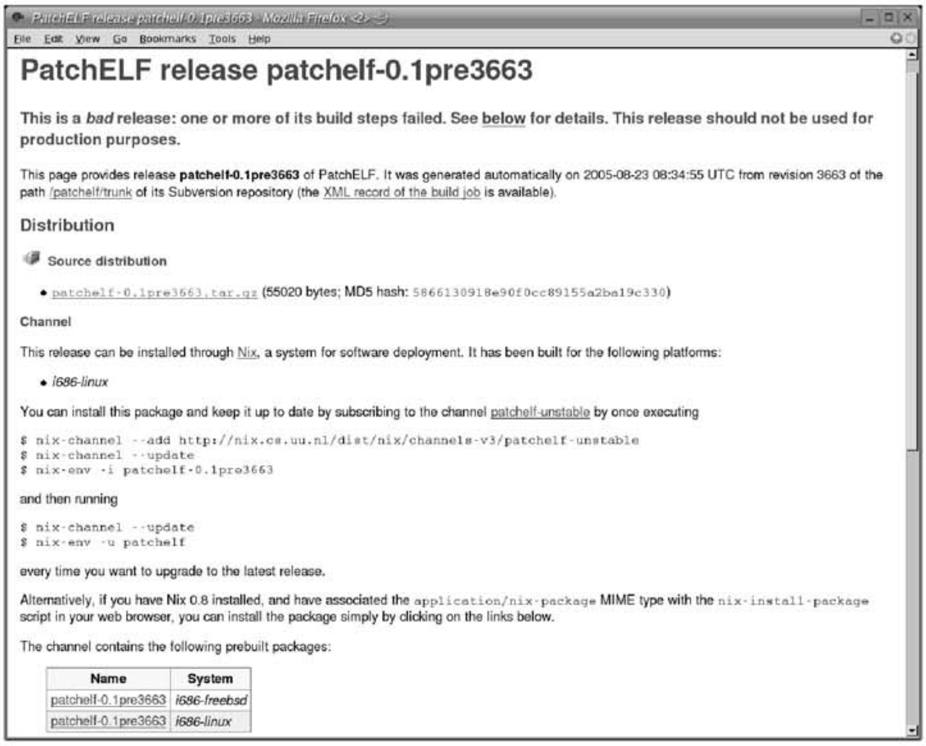
```
inputs: distBaseURL: [7]

with (import ../..) inputs.nixpkgs.path; [8]

rec {

  patchelfTarball = input: svnToSourceTarball "patchelf" input { [9]
    inherit (pkgs) stdenv fetchsvn;
    buildInputs = [ pkgs.autoconf pkgs.automake ];
  };

  patchelfNixBuild = input: pkgs: nixBuild (patchelfTarball input) { [10]
    inherit (pkgs) stdenv;
  };

  patchelfRelease = input: makeReleasePage { [11]
    fullName = "PatchELF";
    contactEmail = "eelco@cs.uu.nl";
    sourceTarball = patchelfTarball input;
    nixBuilds = [
      (patchelfNixBuild input pkgsLinux)
    ];
    inherit distBaseURL;
  };

  patchelfHeadRelease = patchelfRelease (inputs.patchelfHead); [12]
}
```

Figure 3: Build farm Nix expression for PatchELF

after it has failed previously) to the address specified in the job $\boxed{6}$. To prevent a flood of repeated e-mail messages for a failing build, after a job fails, the supervisor will not schedule it again until a certain time interval has passed. This interval increases on every failure using a binary exponential back-off method.

Note that the supervisor is completely Nix-agnostic: it does not care how jobs are performed. It is up to the job script to perform the build in some arbitrary way.

The job script build+upload.sh, on the other hand, uses Nix to perform a build. It instantiates and builds a Nix expression specified as a command-line argument (e.g., ./jobs/nix/patchelf.nix at $\boxed{3}$). This Nix expression is a function that takes as arguments the paths of the inputs declared in the XML job declaration (i.e., at $\boxed{1}$), and the target URL of the release page (specified at $\boxed{5}$). This function must return a *derivation*, which is Nix terminology for a component build action. This derivation is expected to produce a *release page*, which is an HTML page describing the release, plus an arbitrary set of files associated with the release (such as source or binary distributions, manuals to be placed online, and so on).


PatchELF release patchelf-0.1pre3663 - Mozilla Firefox <2>
File Edit View Go Bookmarks Tools Help

PatchELF release patchelf-0.1pre3663

This is a *bad* release: one or more of its build steps failed. See below for details. This release should not be used for production purposes.

This page provides release **patchelf-0.1pre3663** of PatchELF. It was generated automatically on 2005-08-23 08:34:55 UTC from revision 3663 of the path /patchelf/trunk of its Subversion repository (the XML record of the build job is available).

Distribution

Source distribution

- `patchelf-0.1pre3663.tar.gz` (55020 bytes; MD5 hash: `5866130918e90f0cc89155a2ba19c330`)

Channel

This release can be installed through Nix, a system for software deployment. It has been built for the following platforms:

- *i686-linux*

You can install this package and keep it up to date by subscribing to the channel patchelf-unstable by once executing

```
$ nix-channel --add http://nix.cs.uu.nl/dist/nix/channels-v3/patchelf-unstable
$ nix-channel --update
$ nix-env -i patchelf-0.1pre3663
```

and then running

```
$ nix-channel --update
$ nix-env -u patchelf
```

every time you want to upgrade to the latest release.

Alternatively, if you have Nix 0.8 installed, and have associated the `application/nix-package` MIME type with the `nix-install-package` script in your web browser, you can install the package simply by clicking on the links below.

The channel contains the following prebuilt packages:

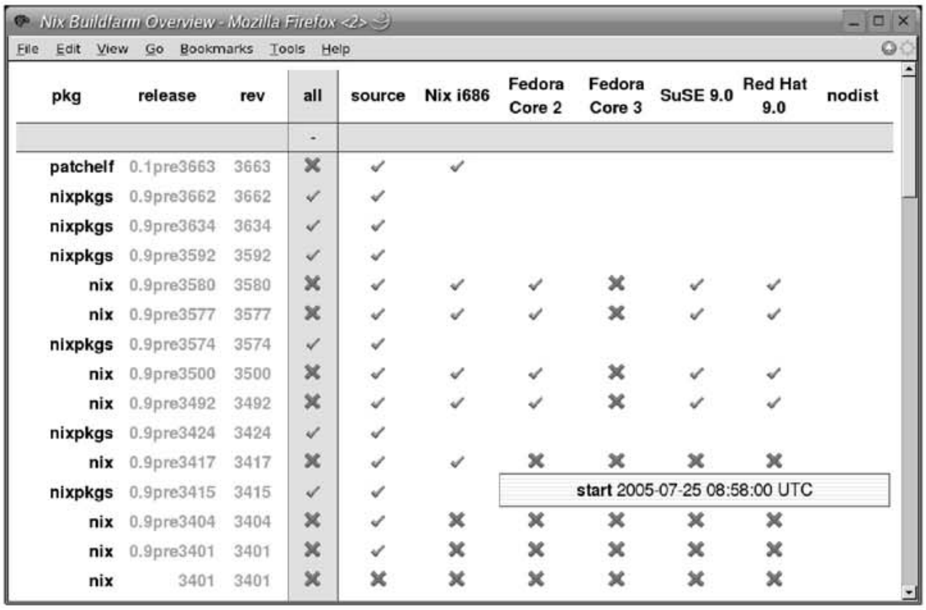
Name	System
patchelf-0.1pre3663	*i686-freebsd*
patchelf-0.1pre3663	*i686-linux*

Figure 4: Release page for PatchELF

Figure 3 shows the Nix expression ./jobs/nix/patchelf.nix that builds a release for PatchELF. (For details on the Nix expression language, the reader is referred to [3, 4].) Release pages are produced by the function patchelfRelease $\boxed{11}$ that accepts a single argument input that points to the component's source code as obtained by the script build+upload.sh by performing a checkout from PatchELF's Subversion repository. A release page produced by patchelfRelease is shown in Figure 4.

The actual production of the release page is done by the generic release page builder function makeReleasePage (brought into scope in the with-expression at $\boxed{8}$). It accepts many arguments, only some of which are shown in the PatchELF example:

- The name of the component (e.g., "PatchELF").
- A contact e-mail address placed on the generated release pages.
- The target URL (e.g., distBaseURL) of the release page. The release page builder does not perform uploads itself (since that is impure) but it needs the target URL for self-references in the release page.


Nix Buildfarm Overview - Mozilla Firefox <2>
File Edit View Go Bookmarks Tools Help

pkg	release	rev	all	source	Nix i686	Fedora Core 2	Fedora Core 3	SuSE 9.0	Red Hat 9.0	nodist
			-							
patchelf	0.1pre3663	3663	✗	✓	✓					
nixpkgs	0.9pre3662	3662	✓	✓						
nixpkgs	0.9pre3634	3634	✓	✓						
nixpkgs	0.9pre3592	3592	✓	✓						
nix	0.9pre3580	3580	✗	✓	✓	✓	✗	✓	✓	
nix	0.9pre3577	3577	✗	✓	✓	✓	✗	✓	✓	
nixpkgs	0.9pre3574	3574	✓	✓						
nix	0.9pre3500	3500	✗	✓	✓	✓	✗	✓	✓	
nix	0.9pre3492	3492	✗	✓	✓	✓	✗	✓	✓	
nixpkgs	0.9pre3424	3424	✓	✓						
nix	0.9pre3417	3417	✗	✓	✓	✗	✗	✗	✗	
nixpkgs	0.9pre3415	3415	✓	✓						
nix	0.9pre3404	3404	✗	✓	✗	✗	✗	✗	✗	
nix	0.9pre3401	3401	✗	✓	✗	✗	✗	✗	✗	
nix	3401	3401	✗	✗	✗	✗	✗	✗	✗	

start 2005-07-25 08:58:00 UTC

Figure 5: Release overview

- A derivation that builds a source distribution (sourceTarball), e.g., the file patchelf-0.1pre3663.tar.gz that can be downloaded, compiled, and installed by users. The source distribution is produced from the Subversion sources (i.e., input) by the function patchelfTarball [9]. (A tarball is a Unix colloquialism for a source distribution.) Here too the actual work is done by an external generic function svnToSourceTarball.

- A list of derivations that perform normal builds of the component from the source distribution (nixBuilds). These are used to automatically populate a *channel* to which users can subscribe, and to generate packages that can be installed directly from the web page by clicking on them ("one-click installations"). In this case the builds are produced by the function patchelfNixBuild [10], which in turn uses the (poorly named) generic function nixBuild.

- Similarly, makeReleasePage accepts attributes for the component's manual, coverage analysis builds, and RPM packages.

The top-level derivation is produced by evaluation of the value patchelfHeadRelease [12]. The name of this attribute was specified in the XML job description at [4]. Building of this derivation will produce the release page and all the distributions included on the release page (in the example, a source distribution and a Nix channel distribution).

The build+upload.sh script, as its name implies, not only builds the derivation but also uploads the release page to the server. Each release is stored under its own URL, e.g., `http://nix.cs.uu.nl/dist/nix/patchelf-0.1pre3663/`. It also performs a nix-push to build and upload the Nix expressions in the channels provided by the release. The uploading of the release is assisted by a server-side CGI script that stores the uploaded files and, when the upload is done, updates various index pages listing the release. Figure 5 shows the automatically generated index of the most recent releases, concisely showing the extent to which each release succeeded.

Reproducing releases An important configuration management property is the ability to reproduce releases in the future. E.g., when we need to fix a bug in some old release of a component, we need to be able to reproduce the entire build environment, including compilers, libraries, and so on. So it is important that we have a record that describes exactly what inputs went into a release.

Therefore the build farm stores a file job.xml as part of every release, e.g., under `http://nix.cs.uu.nl/dist/nix/patchelf-0.1pre3663/job.xml`. This file is the same as the XML job description that went into the supervisor (e.g., the one in Figure 2), except that each input element that referred to a non-constant input such as a HEAD revision has been "absolutised". For instance, the patchelfHead input element has been changed into:

```
<input id='patchelfHead' type='svn'
  url='https://.../repos/trace/patchelf/trunk
  rev='3663' hash='b252b5740a0d...' />
```

That is, it no longer refers to the HEAD revision of the Subversion repository of PatchELF, but to a specific revision. Since this job.xml is a perfectly valid build job, we can feed it into the supervisor to reproduce the build.

Release process As can be seen in Figure 5, each release has a symbolic name, such as patchelf-0.1pre3663. The current build farm implements the policy that names including the string pre in the version string are "unstable" releases (i.e., only intended for developers or bleeding edge users), and are "stable" otherwise. Stable releases are intended to be kept indefinitely, while unstable releases can be deleted eventually. More importantly, stable and unstable releases appear in separate channels. For instance, the URL of the channel for stable PatchELF releases is

```
http://nix.cs.uu.nl/dist/nix/channels/patchelf-stable
```

while the channel for unstable releases is

```
http://nix.cs.uu.nl/dist/nix/channels/patchelf-unstable
```

The release name is computed by the build jobs themselves. The component name (e.g., patchelf) is generally hard-coded into the build job (e.g., at $\boxed{9}$). The version name is usually computed by taking the version number hard-coded into the source (e.g., 0.1) and appending pre$N$, where N is the revision number of the Subversion repository (e.g., 3663), *if* the release is unstable. Whether the release is stable or unstable is also hard-coded in the sources.

For instance, for Autoconf-based components, the release name is usually computed by the configure script, which contains a line

```
STABLE=0
```

in the sources in the main development branch of the project (trunk in Subversion terminology [9]). Thus all releases built from the main branch will be unstable releases. To build a stable release, it suffices to change the line to

```
STABLE=1
```

and rebuild.

In practice, a more controlled process is used to build stable releases. To build a stable release, e.g., patchelf-0.1, the development branch is copied to a special *release branch*, e.g., branches/patchelf-0.1-release. In this branch, the stable flag is set. A one-time job for this branch is then added to jobs.conf. After the release succeeds, the release branch is tagged and removed. That is, branches/X-release is moved to tags/X; e.g., branches/patchelf-0.1-release is moved to tags/patchelf-0.1.

5 Discussion and related work

This section describes some of the advantages and disadvantages of the Nix build farm relative to other continuous integration tools.

The main advantage is the use of Nix expressions to describe and build jobs. It makes the management of the build environment (i.e., dependencies) quite easy and scalable. This aspect is completely ignored by tools such as CruiseControl [10] and Tinderbox [6], which expect the environment to be managed by the machine's administrator. Anthill [11] has the notion of "dependency groups" that allows an ordering between build jobs.

Most of these systems are targeted at testing, not producing releases. Sisyphus [12] on the other hand is a continuous integration system that is explicitly intended to support deployment of upgrades to clients. It uses a destructive update model, which makes it easy to use with existing deployment tools, but bars side-by-side versioning and rollbacks.

The centralised view of the build job for a release is also a big plus. Systems such as Tinderbox have a more "anarchistic" approach: build farm machines perform jobs essentially independently, and send the results to a central machine that presents them on a web page. This is fine for continuous integration per se, but is not a good model if we want integration with release management. Since each build independently selects which revision to build, there is no guarantee that any particular revision will always be built by all machines. Thus there is no guarantee that a complete release page will ever be made.

A fundamental downside to the Nix build farm is that by building in Nix, by definition we are building in a way that differs from the "native" method for the platform. If a component builds and passes the tests on powerpc-darwin, we can conclude that the component *can* work on that platform; but we cannot conclude that it will work if a user were to download the source and build using the platform's native tools (e.g., the C compiler provided in /usr/bin). That is, while the build farm builds the component in a Nix environment, most users will use it in a non-Nix environment. This limits the level of portability testing attained by the Nix build farm.

On the other hand, this situation can be improved by simulating the native environment as closely as possible, e.g., by providing the same tool versions. Nevertheless, there is no getting around the fact that the *paths* of those tools differ; they are in the store, not in their native locations in the file system.

However, we can still build "native" binary distributions in some cases. For example, we use User-Mode Linux (UML) [2] to build RPM packages for various platforms. User-Mode Linux is a normal user space program that runs a complete Linux operating system inside a virtual machine. Thus, the builder of the derivation that builds RPMs is entirely pure: it simply runs UML to build the RPM.

Of course, we can also do "native" builds directly in Nix builders if we are willing to accept a level of impurity (e.g., by adding /usr/bin to PATH). We can even test whether the component installs properly to an impure location (such as /usr/local/my-package) if the builder has sufficient permissions. In fact, the latter need not even be impure as long as the following conditions hold:

- Subsequent derivations do not depend on output in impure locations. Thus, the builder should *remove* the impure output at the end of the build script.

- Locking should be used to prevent multiple derivations from installing into the same impure location. E.g., derivations that want to install into /usr/local/my-package can acquire an exclusive lock on a lock /usr/local/my-package.lock.

A more transient limitation of the prototype build farm is its poor scheduling. (A better supervisor is currently being implemented.) It simply runs jobs after each other in a continuous loop. It supports none of the more advanced scheduling methods discussed earlier. It also does not run jobs in parallel, which leads to poor utilisation of the build farm. For instance, builds on Macs generally take (much) longer than on other machines. A multi-platform job can therefore cause many machines to be idle, as the job waits for the Mac build to finish. It is then useful to start a new job to put the idle machines to good use.

6 Future work

The main focus of future research is automatic testing in large configuration spaces. When testing a component with a large amount of variabilities, the build farm should automatically select interesting configurations in order to maximize the amount of useful knowledge that it produces for the developers. For instance, if a certain configuration succeeds and another does not, the build farm should explore the configuration space, building different configurations, to discover which parameter causes the failure. Similarly, if a certain configuration fails while it did not previously, the build farm should try to isolate the specific commit that introduced the fault.

Another interesting direction is to discover possibly troublesome configurations using source code analysis. For instance, nested #ifdefs in C programs conditional on options in the configuration space may indicate a potential feature interaction that must be tested specifically.

7 Conclusion

A build farm is an indispensable tool in any software development project, but existing implementations have serious limitations: they are hard to maintain because the build environment is not managed, have poor reproducibility, and have no explicit support for building variants of systems. The Nix-based build farm, by virtue of its purely functional component description language, solves these issues. However, much research remains to be done regarding the automatic exploration of large configuration spaces.

The releases of the Nix build farm at Utrecht University are online at `http://nix.cs.uu.nl/dist/`.

Acknowledgements This research was supported by CIBIT and NWO/-JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*. We wish to thank Martin Bravenboer, Armijn Hemel and Merijn de Jonge for their work on implementing the build farm.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] Jeff Dike. A user-mode port of the Linux kernel. In *4th Annual Linux Showcase & Conference (ALS 2000)*, October 2000.

[3] Eelco Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht University, The Netherlands, January 2006. `http://www.cs.uu.nl/~eelco/pubs/phd-thesis.pdf`.

[4] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In Lee Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.

[5] Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th*

International Conference on Software Engineering (ICSE 2004), pages 583–592. IEEE Computer Society, May 2004.

[6] Mozilla Foundation. Tinderbox. `http://www.mozilla.org/tinderbox.html`, 2005.

[7] Martin Fowler and Matthew Foemmel. Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`. Accessed 11 August 2005.

[8] Armijn Hemel. Using buildfarms to improve code. In *UKUUG Linux 2003 Conference*, August 2003.

[9] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion.* O'Reilly, June 2004.

[10] ThoughtWorks. Cruise Control. `http://cruisecontrol.sourceforge.net/`, 2005.

[11] Urbancode. Anthill. `http://www.urbancode.com/projects/anthill/default.jsp`, 2005. Accessed 21 August 2005.

[12] Tijs van der Storm. Continuous release and upgrade of component-based software. In *12th International Workshop on Software Configuration Management (SCM-12)*, September 2005.

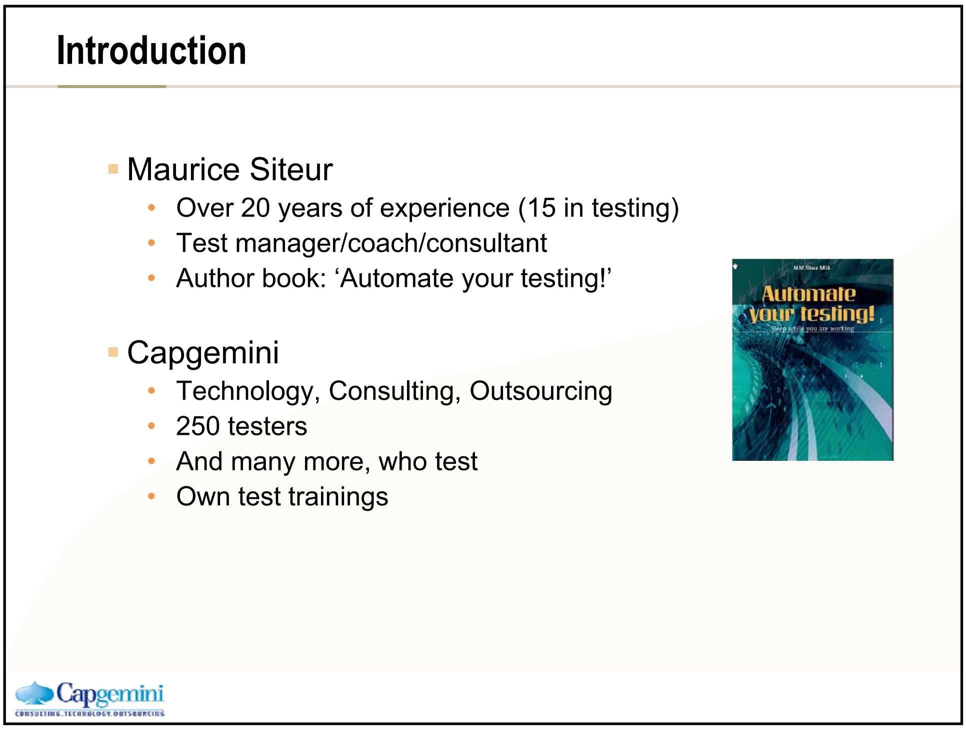
[13] Eelco Visser. Program transformation with Stratego/XT: Strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Software conversions need to be tested

VVSS, 23 march 2007

Maurice Siteur

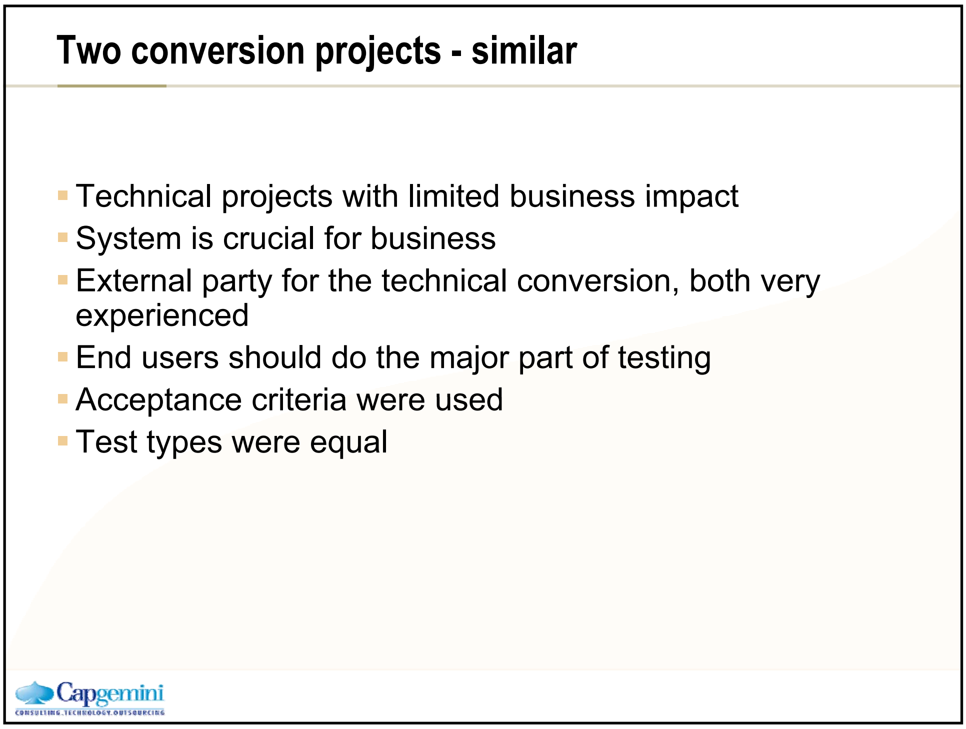


Introduction

- Maurice Siteur
  - Over 20 years of experience (15 in testing)
  - Test manager/coach/consultant
  - Author book: ‘Automate your testing!’
- Capgemini
  - Technology, Consulting, Outsourcing
  - 250 testers
  - And many more, who test
  - Own test trainings

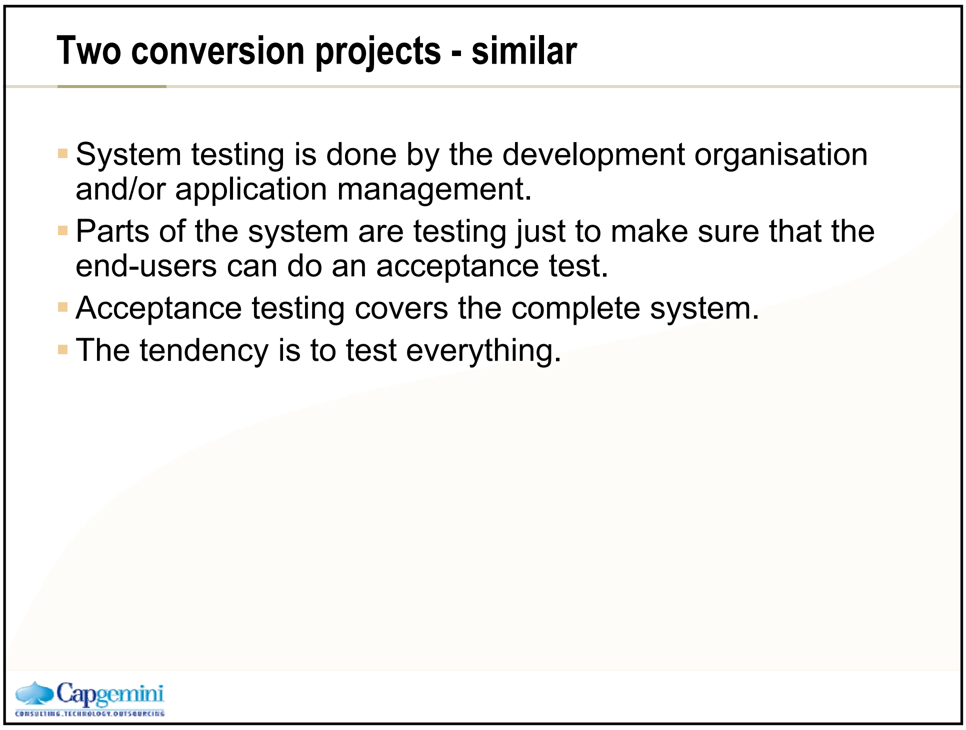
M.M. Siteur MIA
Automate your testing!

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



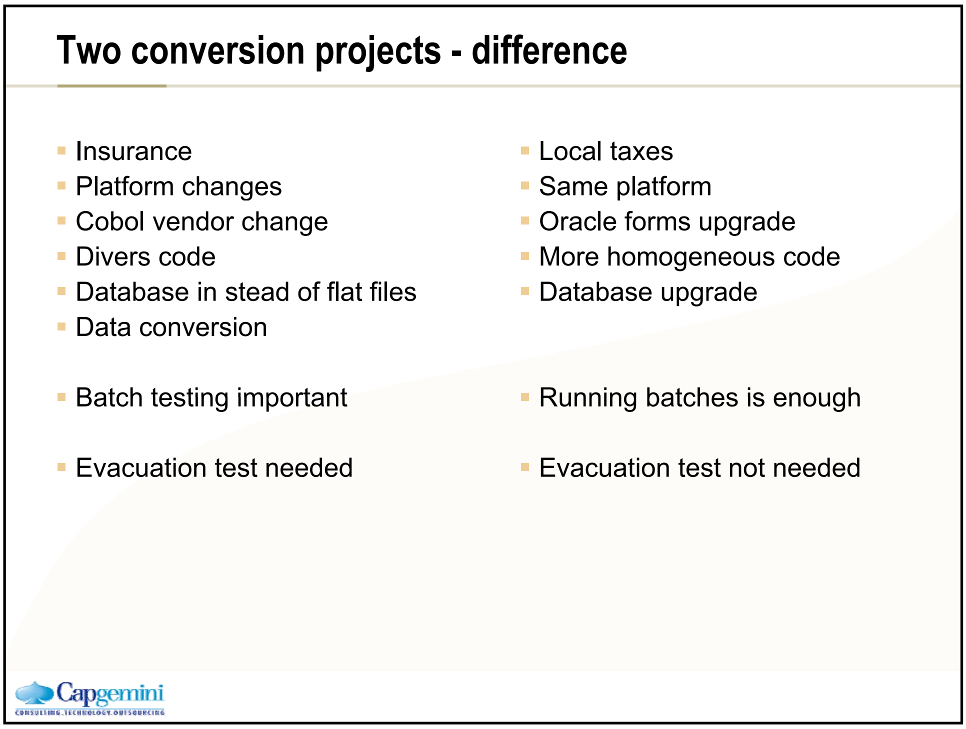
Two conversion projects - similar

- Technical projects with limited business impact
- System is crucial for business
- External party for the technical conversion, both very experienced
- End users should do the major part of testing
- Acceptance criteria were used
- Test types were equal



Two conversion projects - similar

- System testing is done by the development organisation and/or application management.
- Parts of the system are testing just to make sure that the end-users can do an acceptance test.
- Acceptance testing covers the complete system.
- The tendency is to test everything.



Two conversion projects - difference

- Insurance
- Platform changes
- Cobol vendor change
- Divers code
- Database in stead of flat files
- Data conversion

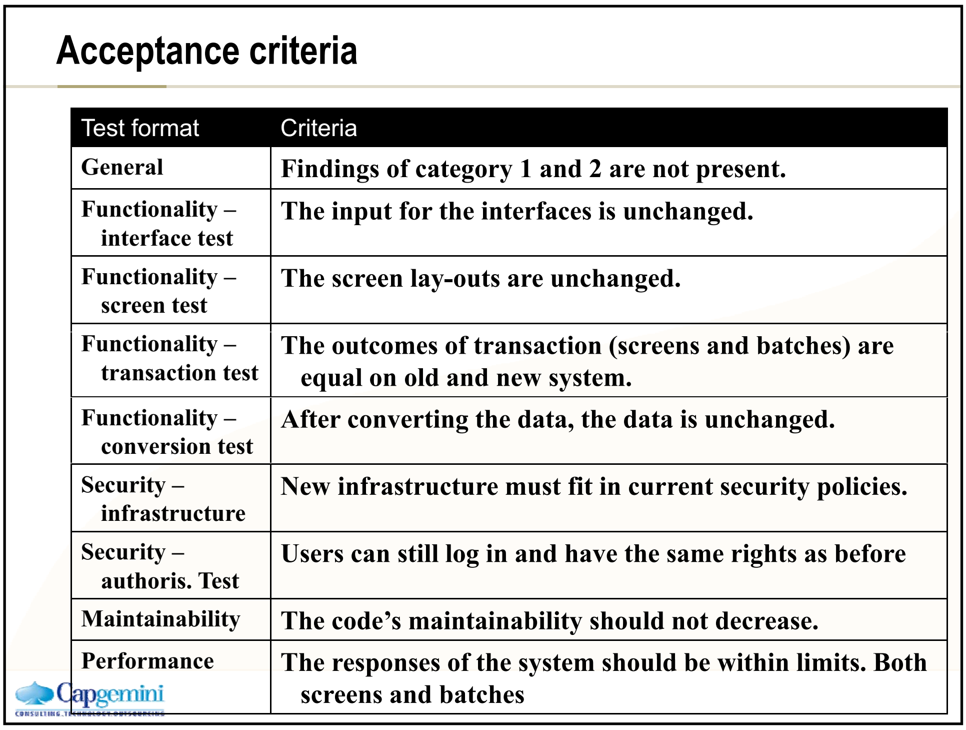
- Batch testing important

- Evacuation test needed

- Local taxes
- Same platform
- Oracle forms upgrade
- More homogeneous code
- Database upgrade

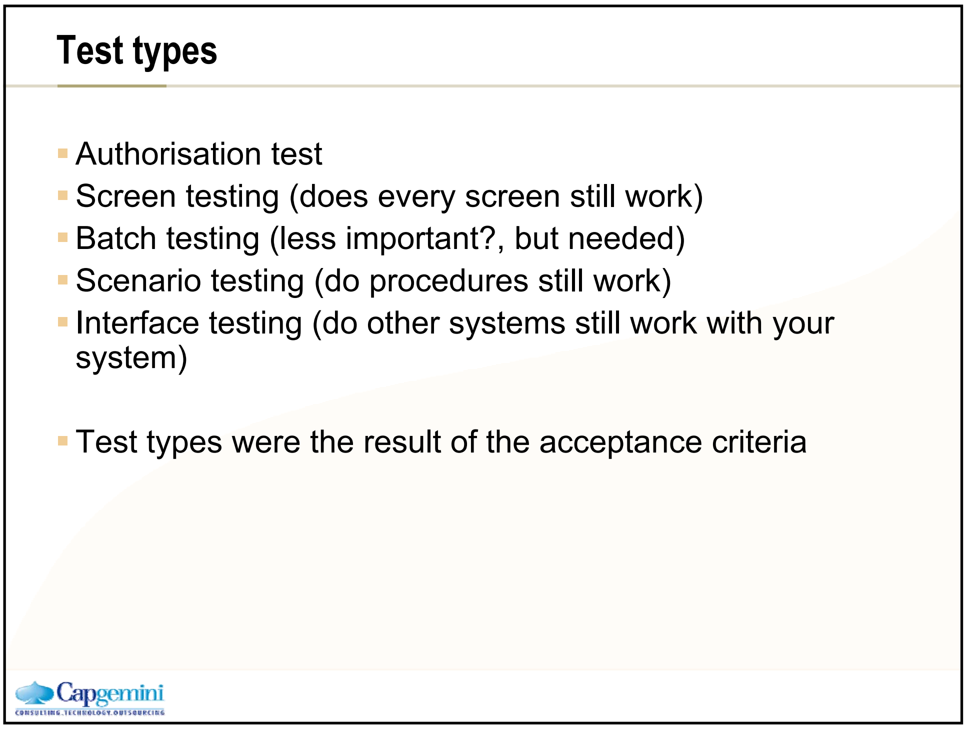
- Running batches is enough

- Evacuation test not needed



Acceptance criteria

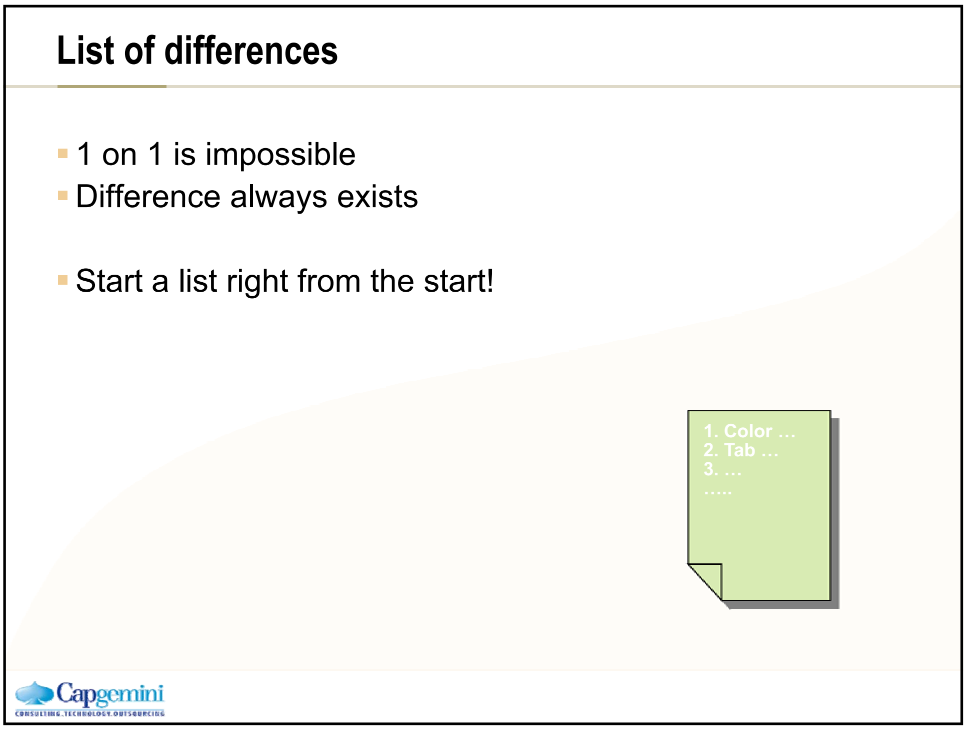
Test format	Criteria
General	Findings of category 1 and 2 are not present.
Functionality – interface test	The input for the interfaces is unchanged.
Functionality – screen test	The screen lay-outs are unchanged.
Functionality – transaction test	The outcomes of transaction (screens and batches) are equal on old and new system.
Functionality – conversion test	After converting the data, the data is unchanged.
Security – infrastructure	New infrastructure must fit in current security policies.
Security – authoris. Test	Users can still log in and have the same rights as before
Maintainability	The code’s maintainability should not decrease.
Performance	The responses of the system should be within limits. Both screens and batches



Test types

- Authorisation test
- Screen testing (does every screen still work)
- Batch testing (less important?, but needed)
- Scenario testing (do procedures still work)
- Interface testing (do other systems still work with your system)

- Test types were the result of the acceptance criteria

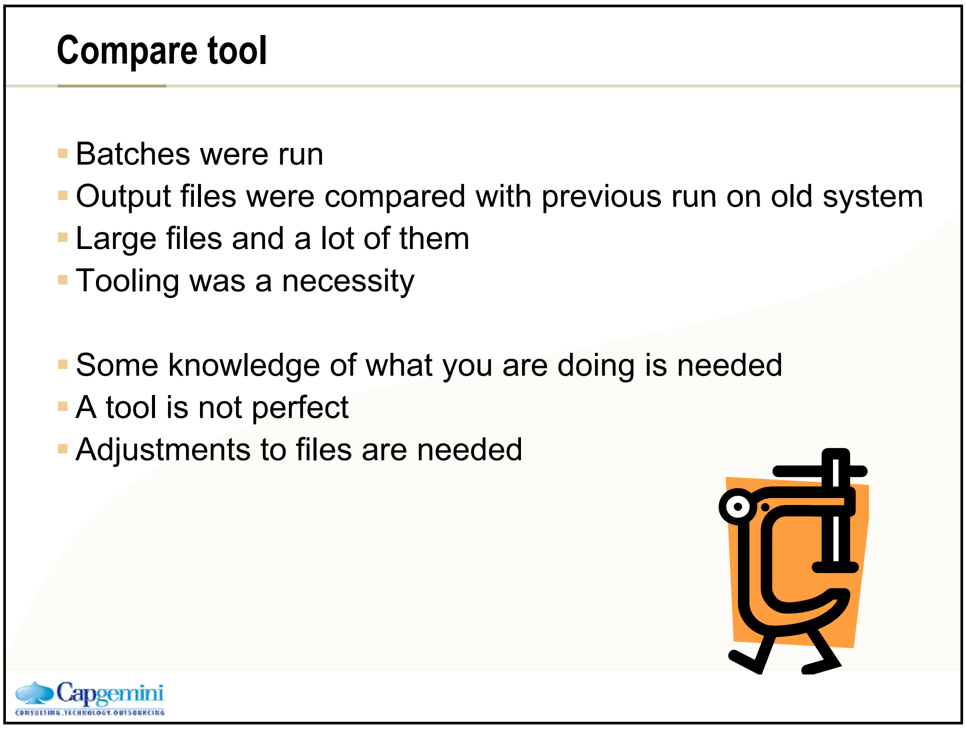


List of differences

- 1 on 1 is impossible
- Difference always exists

- Start a list right from the start!

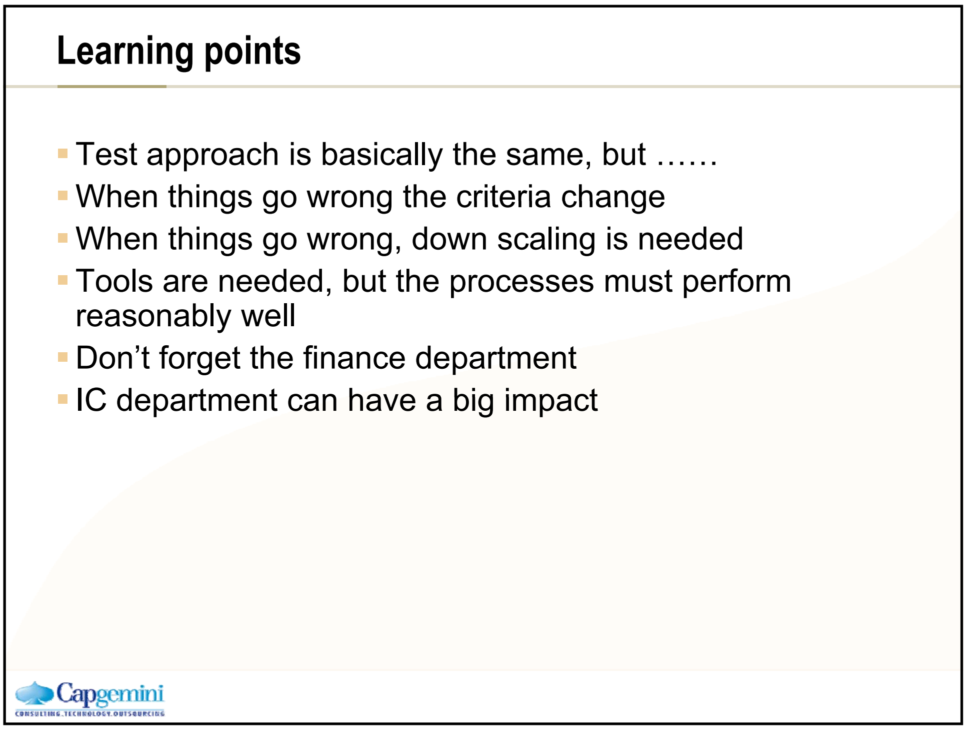
1. Color …
2. Tab …
3. …
…..



Compare tool

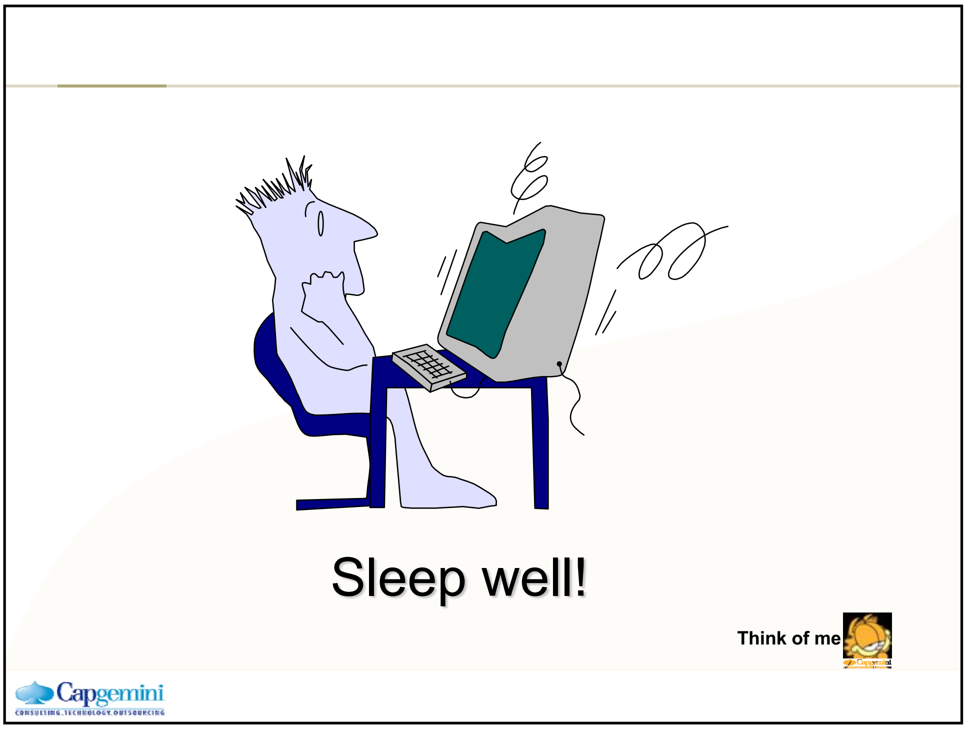
- Batches were run
- Output files were compared with previous run on old system
- Large files and a lot of them
- Tooling was a necessity

- Some knowledge of what you are doing is needed
- A tool is not perfect
- Adjustments to files are needed



Learning points

- Test approach is basically the same, but ……
- When things go wrong the criteria change
- When things go wrong, down scaling is needed
- Tools are needed, but the processes must perform reasonably well
- Don’t forget the finance department
- IC department can have a big impact



Sleep well!

Think of me

An Object-Oriented Framework for Explicit-State Model Checking

Mark Kattenbelt[1] and **Theo C. Ruys**[2] and **Arend Rensink**[2]

Abstract. This paper presents a conceptual architecture for an object-oriented framework to support the development of formal verification tools (i.e. model checkers). The objective of the architecture is to support the reuse of algorithms and to encourage a modular design of tools. The conceptual framework is accompanied by a C++ implementation which provides reusable algorithms for the simulation and verification of explicit-state models as well as a model representation for simple models based on guard-based process descriptions. The framework has been successfully used to develop a model checker for a subset of PROMELA.

1 INTRODUCTION

Model checking is the application of an automated process to formally verify whether a *model* conforms to a *specification* [7, 3]. There are numerous ways in which one could express a model, but typically the model can be interpreted as some sort of automaton. The level of abstraction that is used to describe models in tools varies significantly depending on the model checker, and ranges from low-level automata-based representations (such as the timed automata in UPPAAL [1]) to high-level specification languages that resemble programming languages (such as BIR in Bogor [12]). The specification can also be expressed in various ways, but is usually formulated in terms of properties in some type of temporal logic. The nature of the verification process used in model checkers is heavily dependent on the types of models and specifications it can verify.

Most model checkers are very specialised, and support only a single type of model. Additionally, it is not uncommon for model checkers to introduce their own specification language. Although this specialisation enables tools to optimise their verification algorithms, it does not encourage a reusable design. In order to reuse the functionality contained within model checkers one often has to resort to using the model specification language prescribed by this model checker. As a result, many transformations between input languages of tools currently exist and interaction between tools can only be achieved with considerable effort.

To emphasise the need for reuse, consider the great advancements of model checking in recent years [6]. The aspiration to apply model checking to systems of an industrial scale has led to the introduction of many new complex techniques and algorithms (i.e. partial-order reduction, symmetry reduction, predicate abstraction, slicing algorithms). Implementing a state-of-the-art model checker is not a trivial task, and therefore any opportunity to reuse functionality should be considered beneficial.

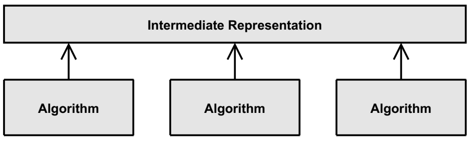


Figure 1 – Model checking frameworks usually have a single intermediate representation. In order to use the framework the model under consideration will have to be expressed in this intermediate representation.

The need for reuse and interoperability has been acknowledged by several others. For example, the *model-checking framework* BOGOR [12], the IF TOOLSET [4], the MODEL-CHECKING KIT [20] and the NCSU CONCURRENCY WORKBENCH [8] all offer a framework to enable reuse in verfication tools, and often employ a layered architecture. Similar to modern compiler suites, most of these frameworks use an *intermediate representation* to which high-level models are translated (see Figure 1). This representation can be a textual description in a modelling specification language, or a programmatic representation. For the frameworks mentioned previously, these intermediate representations are BIR, IF *specification*, *1-Safe Petri Nets* and *Labelled Transition Systems*, respectively.

The verification functionality of these frameworks is realised by algorithms that use this intermediate representation directly. Having a single intermediate representation is advantageous for the optimisation of verification algorithms. However, a drawback of this approach is that the applicability of the framework is limited by the expressiveness of the intermediate representation. Furthermore, a transformation of models to this intermediate representation is not always optimal. The verification algorithms cannot be reused for anything other than the intermediate representation used in the framework.

We have developed a framework that is not limited by a single intermediate representation. We provide a means of describing algorithms such that they can be used by many different intermediate representations. Related to our approach is the MÖBIUS MODELLING ENVIRONMENT [9, 11], which uses the same principle for performance analysis of stochastic models.

The goal of our framework is to enable the development of generic functionality that can be used in several verification tools directly, not necessarily limited to model checkers, and to improve the interoperability of tools. In the remainder of this article we will describe the core essentials of this framework. Details can be found in [16]. The meaning of 'framework' is two-fold in this article:

- *Conceptual architecture.* A conceptual architecture for a model checking framework which enables reuse of code. This architecture enables us to define algorithms that can be reused for different

[1] School of Computer Science, University of Birmingham, United Kingdom. `http://www.cs.bham.ac.uk/~mxk/`. (Supported by EPSRC grant EP/D07956X/1 during the authoring of this extended abstract.)

[2] Formal Methods and Tools group, Faculty of EEMCS, University of Twente, The Netherlands. `http://fmt.cs.utwente.nl/`.

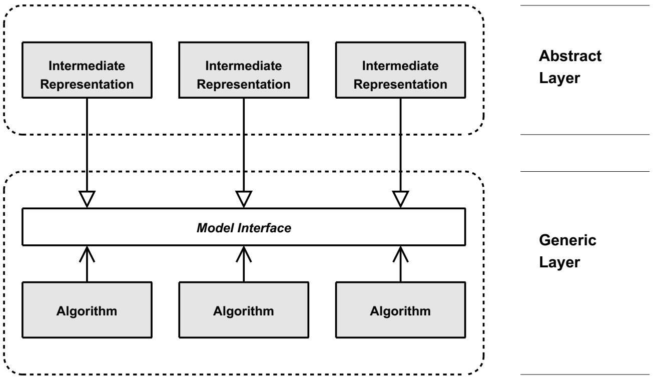

Intermediate Representation
Intermediate Representation
Intermediate Representation
Abstract Layer
Model Interface
Algorithm
Algorithm
Algorithm
Generic Layer

Figure 2 – The conceptual architecture of the framework, divided into a *generic layer* and an *abstract layer*.

intermediate representations. In Section 2 we will introduce this architecture.

- *Concrete architecture.* A proof-of-concept implementation of the conceptual architecture. On a low level, it consists of reusable algorithms for explicit-state verification techniques. On a high level, it provides a graph-based intermediate representation which represents models with guard-based process descriptions. This library is introduced in Section 3.

A proof-of-concept tool is built on top of our concrete architecture and is capable of verifying PROM$^+$ (a subset of PROMELA, see Section 3.4). It combines our intermediate representation with our reusable verification algorithms to realise its functionality.

2 CONCEPTUAL ARCHITECTURE

The conceptual architecture should enable reusable algorithms to be defined over multiple intermediate representations. Our architecture is based on a layered design as depicted in Figure 2, similar to other frameworks. In contrast to other frameworks, algorithms do not refer to the intermediate representation directly (Figure 1), but refer to a model interface instead. We distinguish two layers, a *generic layer* and an *abstract layer*.

Note that we use a slightly informal notation in our architectural diagrams. In general, white blocks are interfaces, whereas grey blocks actually contain some sort of implementation. *Associations* and *specialisation* relationships between blocks are shown using the notation commonly used in UML class diagrams.

2.1 Generic layer

The generic layer contains reusable *algorithms*, as well as a *model interface*. This model interface defines a number of operations to facilitate the algorithms. Additionally, we abstract from the types that are used in the model interface by means of type parametrisation (e.g. generics in JAVA, templates in C++).

The idea is that the model interface abstracts over the most elementary types used in the algorithms, which are likely to be different for different intermediate representations. In this way the algorithms need not to be concerned with the implementation of these types, and intermediate representations can provide their own custom implementation of these types. The model interface defines operations over these types such that the algorithms can efficiently realise their functionality using these operations, but it is the intermediate representations that actually implement these operations.

The most obvious choice of a generic layer would be one to facilitate *explicit-state model checking*. In this type of model checking each state is explicitly represented, and the verification process can usually be reduced to some type of exhaustive search over the state space. Candidate types for type parameters are elementary types such as *states* and *transitions*, whereas operations are likely to facilitate the on-the-fly construction of the state space (i.e. an operation to retrieve successors of a state). A generic layer for explicit-state model checking is discussed in Section 3. Other possible generic layers could facilitate *symbolic* or *bounded model checking*, where candidates for type parameters would include *sets of states* or *clauses*, respectively [16].

Generally speaking, anything contained within the generic layer is meant for use with any intermediate representation, and therefore uses type parameters. Items in the abstract layer are specific to an intermediate representation and therefore do not apply type parameters. Any specialisation relationship between the generic and the abstract layer therefore also implies a specialisation of types.

2.2 Abstract layer

The abstract layer contains intermediate representations of a programmatic form. The basic idea is that such an intermediate representation *specialises* the model interface in a generic layer. In other words, an intermediate representation implements the operations of the model interface for a particular set of types. In the context of explicit-state model checking, intermediate representations in the abstract layer can be very diverse, ranging from 'low-level' representations such as Labelled Transition Systems (LTS), and Graph Transition Systems (GTS) [18] to 'high-level' representations such as Process Algebras (PA) or those used in SPIN [13] and BOGOR [19].

The benefit of using type parameters is that an intermediate representation can implement its own elementary types. For instance, an intermediate representation that implements a model interface of a generic layer for explicit-state model checking can define its own state type. This is useful because the information contained within a state is significantly different for different intermediate representations. For instance, the information contained within a state of a PA model is very different from the state of a PROMELA model. In terms of an intermediate representation of an abstract layer for symbolic model checking, this type specialisation could be used to implement different ways of representing a *set of states*, such as BDDs [17, 5] or MDDs [15]. Arguably, the same effect can be accomplished with subtyping, but this introduces more flexibility (and overhead) than is necessary. The MÖBIUS tool uses a similar approach, and applies subtyping [10] as well as type parametrisation [11].

An alternative conceptual architecture is employed in the NCSU CONCURRENCY WORKBENCH [8]. In this framework intermediate representations can be translated into a LTS automatically by using the Structured Operational Semantics (SOS) of these intermediate representations.

3 CONCRETE ARCHITECTURE

In Figure 3 an overview of our library is shown. The generic layer consists of an explicit-state model interface and algorithms for simulation and verification. The motivation for this library originated from the desire to offer a modular alternative to state-of-the-art tool SPIN [13], which is reflected in the abstract layer. The 'software model' intermediate interpretation is meant for targeting a subset

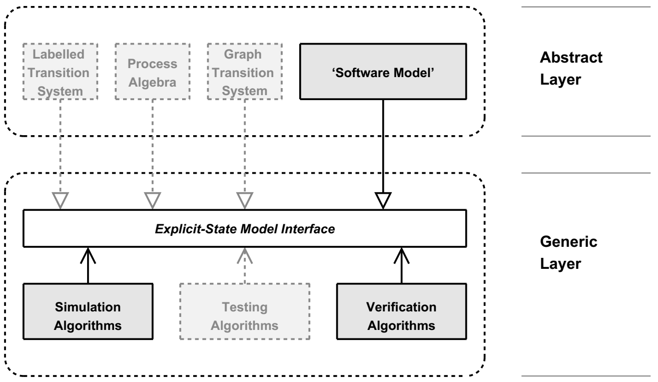


Figure 3 – The concrete architecture of the framework as implemented in our library. Elements that were not implemented, but are shown in the figure to provide a context, have dashed borders.

of PROMELA called PROM$^+$, and is the intermediate representation used in our proof-of-concept model checker. This representation could be extended to support other model specification languages such as BIR, and is therefore not dedicated to a single tool.

The components of the library are written in C++, and feature a modular object-oriented design. Functionality in the generic layer includes simulation and reachability algorithms. The 'software model' intermediate representation comprises the largest part of the library, as it is aimed to be as general as possible.

3.1 Explicit-state model interface

The definition of a model interface has two important features, a set of type parameters and a set of operations. These types and operations should be chosen carefully because all intermediate representation that use this generic layer will have to conform to this interface. Additionally, the operations are to enable all prospective algorithms of this generic layer to realise their functionality efficiently.

The model interface of our prototype can be found in Listing 1. This listing shows that our implementation language is C++. Although it is not necessary to understand C++ in order to understand the principles of our design, we use code samples to illustrate our design. These principles could also be implemented in another language, such as JAVA. We will provide a brief explanation with each code sample, but we refer to [21] for a more concise reference on C++.

Note that we do not define the model interface for any specific type of model representation (e.g. LTS or Kripke structures) but attempt to provide an interface for a large class of automata-based models. In our implementation we chose to abstract from the type of states (`S`), type of labels (`L`), and type of transitions (`T`) used in the intermediate representations. The set of operations is defined such that model information can be retrieved on-the-fly. These functions are abstract (e.g. pure and virtual in C++), and will need to be implemented by any intermediate representation. The initial state object of a model can be retrieved using the `getInitialState` function. Given a state of the model, we can retrieve all outgoing transitions objects of this state in a total order using the `getFirstTransition` and `getNextTransition` functions.

Note that our choice of operations has already limited the type of intermediate representations that can use this generic layer (i.e. precisely *one* initial state is required and all outgoing transitions of a state are required to be in some total order). This is a compromise between generalising the model interface to be compatible with a large number of intermediate representations and providing a set of operations through which explicit-state model checking can be achieved efficiently.

```
template <typename S, typename L, typename T>
class ExplicitStateModelInterface
{
  public:
  virtual S* getInitialState() =0;
  virtual T* getFirstTransition(S* s) =0;
  virtual T* getNextTransition(T* tr) =0;

  virtual S* getSource(T* tr) =0;
  virtual L* getLabel(T* tr) =0;
  virtual S* getTarget(T* tr) =0;
};
```

Listing 1 – The model interface of our library consists of a single C++ class called `ExplicitStateModelInterface`.

To complete the interface we add methods that map transition objects to the source state object (`getSource`), to the target state object (`getTarget`) and to a label object (`getLabel`). All operations are conveniently gathered in the model interface such that there are no restrictions on the implementation of the state, label and transition objects.

Note that all operations work with pointers to elementary types, to facilitate the need for sharing instances. For example, labels are likely to label multiple transitions of the model, and it might be useful for these to be represented by the same label instance.

The prototype implementation of this generic layer actually uses reference counting pointers to keep track of all instances that were provided through the model interface. This arises from the fact that it is written in unmanaged C++, and any created instance will need to be deleted somewhere. As instances might be shared, it is not obvious where this deletion should happen. Reference counting pointers provide additional flexibility to avoid this problem. We use regular pointers in our code listings to make them easier to understand.

3.2 Generic Algorithms

To illustrate how one can define reusable algorithms over the model interface we use the example of a basic depth-first search, as provided in [14]. Although this algorithm is not a very realistic example of an algorithm used in explicit-state model checkers, it is useful to illustrate how this algorithm can be implemented generically (i.e. for all intermediate representations). A more realistic example can be found in [16].

In an idealistic scenario the model interface itself would provide sufficient functionality for any algorithm that we wish to implement in the generic layer. In practice this is not feasible. For example, in the case of our basic depth-first search algorithm, we are looking for erroneous states. As we cannot assume anything about the state type, and this information is not present in the model interface, we will need to get this information elsewhere. Furthermore, instead of simply looking for erroneous states, we would like to generalise the depth-first algorithm to look for any type of 'goal state'. This results in the introduction an additional interface called `GoalCondition`, which contains a single abstract function `isGoalState` that can be used to determine whether a state is a goal state or not (see Listing 2). Note that this interface also uses type parameters, and that if an intermediate representation wishes to use the depth-first algorithm then it will also have to provide an implementation of the

```
template <typename S, typename L, typename T>
class GoalCondition
{
  public:
  virtual bool isGoalState(S* s) =0;
};
```

Listing 2 – The `GoalCondition` interface has a single function `isGoalState` which identifies states of interest. This function is typically implemented in the abstract layer.

`GoalCondition` interface, specialised with the same types (note that type parameters `T` and `L` are not essential for this particular interface, but throughout our implementation we have included all type parameters in all interfaces for consistency).

The definition of the `GoalCondition` enables a search for an arbitrary set of states. This set will typically be specific to an intermediate representation, and therefore will be implemented in the abstract layer. Examples are *accepting states* for automata, *erroneous states* for programs, or the *solved state* for Rubiks cubes. Alternatively, the set of states could be identifiable in a generic way (i.e. for all intermediate representations). As we cannot assume anything about the types of states, transitions and labels, this is not very common. Examples are the *initial state* and *deadlock states*. Deadlocks states can be found generically by checking whether a state has any outgoing transitions.

Now that the issue of detecting erroneous states has been addressed we can implement the basic depth-first algorithm generically. An implementation of this algorithm inside an encapsulating class is shown in Listing 3. This encapsulating class, `BasicDepthFirstSearch`, also abstracts over the type of state, label and transitions used in the algorithm. It has two fields, `m` is an implementation of an `ExplicitStateModelInterface` and `g` is an implementation of `GoalCondition`, both specialised with the paramethised types of the encapsulating class. The `dfs` function is a direct translation the algorithm in [14] to `C++` code, but implemented generically.

If we were to include `BasicDepthFirstSearch` in our architectural diagram, this would result in a generic layer as depicted in Figure 4. The `BasicDepthFirstSearch` block has an association with the `ExplicitStateModelInterface` and with the new interface `GoalCondition`, because these are fields used in the algorithm. The `GoalCondition` has two generic implementations, and is potentially implemented for some intermediate representations in the abstract layer.

The introduction of another interface (`GoalCondition`) does not add significant requirements to the abstract layer. Firstly, implementing an interface other than the model interface should be fairly straight-forward. For example, if there exists an intermediate representation for automata, then checking whether a state is accepting (e.g. implementing a `GoalCondition` for *accepting states*) should be a trivial task. Additionally, these interfaces do not have to be implemented unless the algorithm that uses these interfaces is used. Finally, it is not impossible that there already exists a generic implementation with the desired functionality.

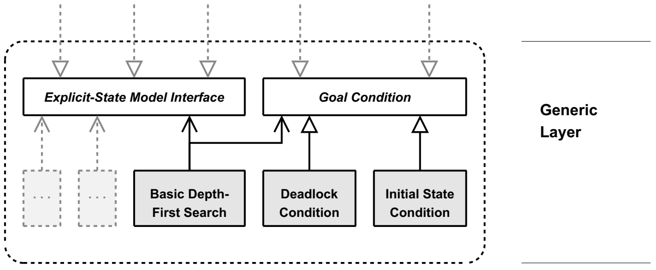


Figure 4 – The generic layer of the framework as it would look if we encorporated `BasicDepthFirstSearch` and `GoalCondition`.

Although we used a very simple example to illustrate the implementation of generic functions in our framework, we argue that this technique is scalable and can be applied to realistic algorithms that are used in model checking today. The actual algorithms implemented in our framework provide both simulation and reachability algorithms. Rather than providing a number of distinctly separate algorithms, we chose to apply a more modular approach. We would like to emphasise that our implementation of simulation and verification functionality is just one of many possible approaches. A simplified overview of the implemented generic layer is presented in Figure 5. As is evident from the figure, algorithms are no longer represented by a single block, but are divided into several blocks to provide a greater degree of flexibility.

The `Simulation` class is associated with both a `SimulationStrategy` and a `SimulationObserver`. These are both interfaces, and can be implemented generically or can be specialised to suit a specific intermediate representation.

```
template <typename S, typename L, typename T>
class BasicDepthFirstSearch
{
  private:
  /* model under consideration */
  ExplicitStateModelInterface<S, L, T>* m;
  /* the goal of this search */
  GoalCondition<S, L, T>* g;

  ...

  public:
  void dfs(std::set<S*>& Statespace, S* s)
  {
    /* if s is a goal state */
    if (g->isGoalState(s)) {
      /* report goal */
    }
    else {
      /* add s to state space */
      Statespace.insert(s);

      /* iterate over transitions of s */
      T* tr = m->getFirstTransition(s);
      while (tr != 0) {

        /* get target state of tr */
        S* t = m->getTarget(tr);

        /* if t is not in Statespace, then dfs */
        if (Statespace.find(t) == Statespace.end())
          dfs(Statespace, t);

        /* get next transition of s*/
        tr = m->getNextTransition(tr);
      }
    }
  }
};
```

Listing 3 – A generic implementation of the basic depth-first search algorithm in [14]. Requires an implementation of an `ExplicitStateModelInterface` and a `GoalCondition`.

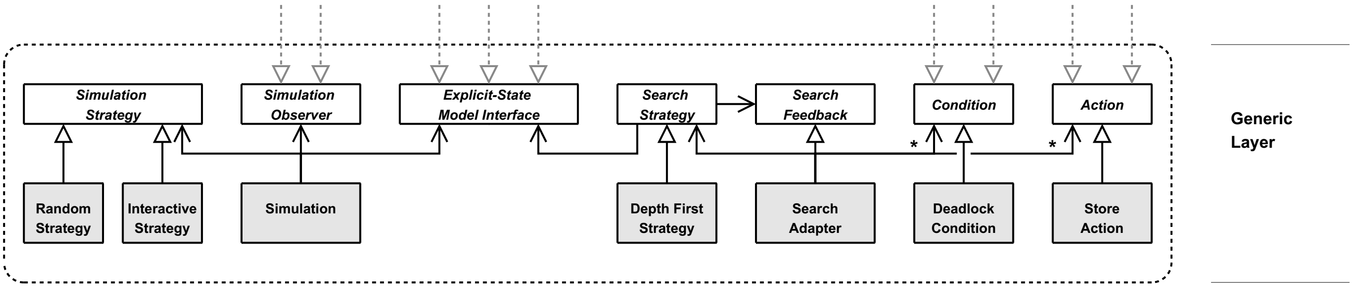


Figure 5 – The architecture of the generic layer, as implemented in our framework. The left-hand side facilitates a simulation algorithm, whereas the right-hand side shows a modular implementation of a reachability algorithm.

The `SimulationStrategy` is responsible for choosing a path through the model, and has generic implementations for random, interactive and guided strategies. Specialised implementations could include random strategies that take into account the probabilities associated with transitions, if it is a specialisation for an intermediate representation that has such a notion of probabilities. The `SimulationObserver` provides a way for tools to observe the simulation, and would most likely consist of specialised implementations to update user interfaces.

The search functionality offered by our framework is slightly more complex. The `SearchStrategy` is an interface for search strategies, whose implementations will have full control over the order of traversal of the states in the model. Currently the only implementation available is a depth-first strategy. Any strategy relies on feedback from `SearchFeedback` such as '*this state was previously visited*', '*this is a new state*' or '*this is a goal state*'. `SearchAdapter` implements this feedback procedure by maintaining pairs of *conditions* and *actions*. `Condition` identifies certain states or transitions, and is in fact very similar to `GoalCondition`. When such a condition holds then an `Action` is executed. Examples of such actions could include storing a state in a store, starting a nested search or reporting a goal state. The feedback given by the `SearchAdapter` is dependent on the actions that were executed. Simple searches can be constructed by combining conditions and actions in a simple fashion, e.g. '*always store a state*' and '*if this state is in the store, report that this state was previously visited*' and '*if this is a goal state, report this goal state*'. The simulation and search functionality of our framework is explained in more detail in [16].

The usage of type parameters in algorithms does not necessarily have an impact on performance. The abstraction is resolved at compile-time, and does not add significant run-time cost in modern compilers. For instance, the standard library of C++ (`std`) is also based on type parametrisation and is generally considered to be very efficient.

3.3 Graph-based intermediate representation

We have explained how generic functionality can be defined in the generic layer, but have not yet addressed any implementation of the abstract layer. In this section we will discuss the intermediate representation that was implemented in our prototype tool. We would like to emphasise that this implementation is only one of many possible intermediate representations that could be defined.

The type of models we will be trying to target are simple software-based models with guard-based process descriptions, global and local variables with primitive and pointer types, as well as dynamic process and data creation. We will use this intermediate representation to verify a subset of PROMELA in Section 3.4. Listing 4 shows that we have a `SoftwareModel` which implements the `ExplicitStateModelInterface` and specialises the type parameters with `SoftwareStates`, `Statements` and `SoftwareTransitions`. The remainder of this section will elaborate on the implementation of `SoftwareModels`.

Due to the dynamic nature of our target models, we will use a graph-based representation of states in our intermediate representation. Our graph-based state representation is based on the representation used in BOGOR [19]. Data values and process instances are *nodes*, whereas variables induce *edges* in our *state graphs*. If a variable has a value then it is represented as an edge originating from the scope in which it is defined (typically a process instance) to the data value this variable evaluates to in the current state of the model. Additionally, we have a global node which acts as the start node for global variables.

We chose to model pointer variables as special kinds of variables, rather than introducing an additional level of indirection. State-graphs annotate edges that are induced by these pointer variables. Typically, pointer variables model heap data, whereas normal variables model stack data. We require heap and stack data values to be strictly separate (i.e. a pointer variable can never point to the value of a normal variable).

By using the state graph representation our intermediate representation is a simplification of real-life software, because we do not model concepts such as memory location, functions and classes. We

```
class SoftwareModel
  : public ExplicitStateModelInterface
  <
    SoftwareState,
    Statement,
    SoftwareTransition
  >
{
  virtual SoftwareState*
    getInitialState();
  virtual SoftwareTransition*
    getFirstTransition(SoftwareState* s);
  virtual SoftwareTransition*
    getNextTransition(SoftwareTransition* tr);

  virtual SoftwareState*
    getSource(SoftwareTransition* tr);
  virtual Statement*
    getLabel(SoftwareTransition* tr);
  virtual SoftwareState*
    getTarget(SoftwareTransition* tr);
};
```

Listing 4 – An implementation of the `ExplicitStateModelInterface` by an intermediate representation of `SoftwareModels`.

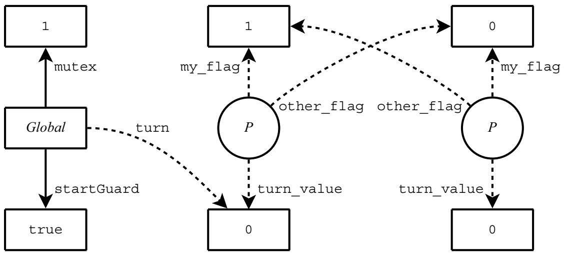


Figure 6 – A graphical representation of a state in our intermediate representation.

consider abstraction over memory locations to be a good thing, as this means detecting heap symmetry reduces to checking whether two state graphs are isomorphic. The other simplifications have been made due to time limitations, and would be welcome additions to our intermediate representation. We informally address the inclusion of features such as functions and classes in [16].

Figure 6 shows a state graph of a model, which is actually a reachable state of the Prom^{+} model shown in Listing 5. The formal definition of state graphs has been explained in [16], we shall just explain them informally. Circles are process instances, whereas rectangles are data instances. Edges induced by variable values are labelled with the variable name and are dashed only if the variable is a pointer variable.

The implementation of state graphs is relatively straight-forward (see the top left portion of Figure 8). A SoftwareState has an association with a GlobalInstance and some ProcessInstances. We presume that every other node in the state graph is reachable from either the global instance or a process instance.

It is clear from the example that the models we try to target are very dynamic in nature. For instance, we cannot determine how many process instances are going to be created during runtime by means of static analysis, nor can we predict what state graphs we will encounter. This implies that it is sensible to construct the state space on-the-fly (alternatively one could construct the whole state space at once, but just feed the model interface this information on-the-fly).

To facilitate the on-the-fly creation of our models, we will need to implement the semantics of our model through our transition and label type. We mentioned previously that a *statement* is a suitable candidate for a label type. As is evident from Listing 5, statements are part of the control-flow of process types. Multiple process instances can share the same process type, and this process type can be shared by multiple SoftwareStates. To facilitate the notion of type, we introduce a type graph to our intermediate representation (which is a type graph for every state graph of the model). This type graph includes nodes for process types, data types, and shows possible variable relations between these types. It is here that we store model-wide information such as the control-flow, the types of variables, statements, etc. This type graph *can* be extracted by means of static analysis. Figure 7 shows the type graph extracted from Listing 5.

The implementation of the type graph is shown on the top right portion of Figure 8. Similarly to the state graphs, all nodes in the type graph are reachable from the GlobalType or a ProcessType. As this information is model-wide, a SoftwareModel has associations with the GlobalType and all ProcessTypes. As can be seen in Figure 8, ProcessTypes implement the model interface too, because their control-flow is considered to be a type of explicit-state model too. This makes it possible to query the control-flow of process types in an on-the-fly manner.

A SoftwareModel normally only has an initial state graph and a type graph at its disposal to realise the operations in the model interface, which are extracted using static analysis. We will informally explain how a SoftwareModel implements the model interface using only this information. The getInitialState is simply a trivial operation to retrieve the initial state. The getFirstTransition and getNextTransition operations are responsible for constructing all enabled SoftwareTransitions originating from a SoftwareState. Although this information is retrieved in several steps, here we will suffice with explaining how one can extract all enabled transitions from a SoftwareState (which is given as an argument) using Figure 8.

The idea is that each SoftwareState contains a certain number of ProcessInstances. Each of these ProcessInstances has a ControlFlowState which represents the program counter of this process. For each of these ProcessInstances, we look up the corresponding

```
byte mutex;
bit * flag_1, * flag_2, * turn_1, * turn_2, * turn;
bool startGuard;

active [0] proctype P(
  bit * my_flag;
  bit * other_flag;
  bit * turn_value)
{
  /* Wait for initialisation */
  startGuard;
  do
  ::*my_flag = 1;
    turn = turn_value;
    (*other_flag == 0 || turn != turn_value);

    /* Begin critical section */
    mutex = mutex + 1;
    mutex = mutex - 1;
    /* End critical section */

    *my_flag = 0;
  od;
}

active [1] proctype Init()
{
  mutex = 0;
  startGuard = false;

  flag_1 = new bit; *flag_1 = 0;
  flag_2 = new bit; *flag_2 = 0;
  turn_1 = new bit; turn = turn_1;
  turn_2 = new bit;

  run P(flag_1, flag_2, turn_1);
  run P(flag_2, flag_1, turn_2);

  /* Do not break symmetry */
  reset flag_1;
  reset flag_2;
  reset turn_1;
  reset turn_2;

  /* Now start! */
  startGuard = true;
}
```

Listing 5 – An implementation of Petersons mutual exclusion algorithm [2] in Prom^{+}.

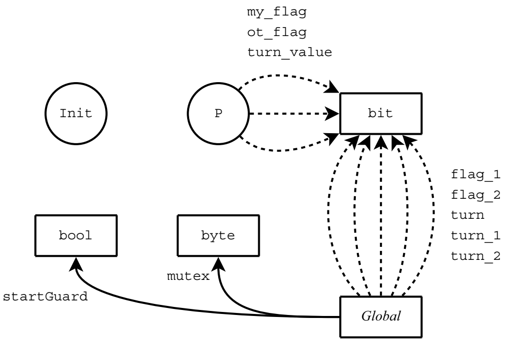


Figure 7 – The type graph of all state graphs in the model described by Listing 5.

ProcessType. Using the getFirstTransition and getNextTransition of the ProcessType we can retrieve all possible ControlFlowTransitions from the current ControlFlowState. An expression in the Statement associated with this transition (i.e. the guard) enables us to see whether this transition is available for the current state. If it is, then we can construct a SoftwareTransition using the information that we have just found. A SoftwareTransition is basically a tuple of the SoftwareState, a reference to the executing ProcessInstance and the ControlFlowTransition that is associated with this step. The getSource and getLabel functions of the model interface are therefore trivial to implement, and if at some point the getTarget function is called, then the ControlFlowTransition along with the Statement are responsible for copying and modifying the SoftwareState into a new state. Statements are therefore basically programmatic implementations of graph morphisms, and contain implementations such as assignments, expressions, assertions etc. Due to the dynamic nature of our states implementing these statements is not entirely straight-forward. For instance, in order to assign to a variable, its edge must first be located in the graph.

In addition to the functionality shown in Figure 8 we have also implemented a means of linearising SoftwareStates to bit vectors. Representing states as a sequence of bits is much more efficient than representing them as a number of object instances and is the typical approach undertaken by model checkers. Our linearisation uses the fact that SoftwareStates are actually programmatic representations of state graphs. By means of a simple algorithm we encode our graphs, using the fact that there exists a type graph, that every node is reachable from the global instance or a process instance, and that the state graphs are deterministic. Details of our encoding algorithm can be found in [16] and is different to the method used in [19]. Heap symmetry is achieved automatically, as isomophic graphs are encoded to the same bit vector with our algorithm. Additionally, process symmetry can be achieved by ignoring the process identifier of process instances during the encoding procedure, and by letting the encoding algorithm look only at the type of the processes and their program counters. This creates representatives that are not necessarily canonical (i.e. some thread-symmetrical states have different representatives), but offers a reasonable reduction with a low run-time overhead.

3.4 PROM^{+} model checker

As a proof of concept, the framework has been used to build a model checker for PROM^{+}, which is a subset of PROMELA [13] augmented with features for dynamic memory allocation. The grammar of PROM^{+} is depicted in Figure 9.

```
       prom ::= (mult_decl ';')* (proctype ';')+
       decl ::= type ('*')? ident
  mult_decl ::= type ('*')? ident ( ',' ('*')? ident )*
   proctype ::= 'active' '[' number ']' 'proctype' ident '(' ( params )? ')'
                '{' (mult_decl ';')* (stmnt ';')+ '}'
     params ::= decl ( ';' decl )*
       type ::= 'bit' | 'bool' | 'byte' | 'short' | 'int'
      stmnt ::= do_stmnt | if_stmnt | assgn_stmnt | new_stmnt | reset_stmnt | run_stmnt |
                expr | assert_stmnt | 'skip'
   do_stmnt ::= 'do' (branch)+ 'od'
   if_stmnt ::= 'if' (branch)+ 'fi'
     branch ::= '::'('else' ';')? (stmnt ';')* ('break' ';')?
assgn_stmnt ::= ('*')? ident '=' expr
  new_stmnt ::= ident '=' 'new' type
reset_stmnt ::= 'reset' ident
  run_stmnt ::= 'run' ident '(' ( args )? ')'
       args ::= expr ( ',' expr )*
       expr ::= expr ( '<' |'<=' | '>' |'>=' | '==' |'!=' | '&&' |'||' | '+' |'-' | '*' |
                '/' | '%' ) expr | ( '!' |'-' ) expr | '(' expr ')' | 'true' | 'false' |
                number | ( * )? ident
assert_stmnt ::= 'assert' '(' expr ')'
      ident ::= ( 'a' | ... | 'z' | 'A' | ... | 'Z' | '_' )+
     number ::= ( '0' | ... | '9' )+
```

Figure 9 – The grammar of PROM^{+} in EBNF style. The syntax and semantics of PROM^{+} are based on PROMELA [13].

The syntax of PROM^{+} is almost all interpretable as PROMELA, and the although PROM^{+} syntax is much more restricted, the syntax that is permitted has the same semantics as in PROMELA. The semantics of PROMELA are described in detail in [13]. In contrast to PROMELA, PROM^{+} only allows the declaration of variables of primitive types, and requires these to be either prior to all process declarations or prior to any statement in a process declaration. There is no means of explicitly giving these variables an initialisation value, and all variables are initialised with 0. Although PROM^{+} lacks many features of PROMELA (i.e. channels, arrays, typedefs, mtypes), it does have facilities for dynamic object creation that PROMELA does not have. We will briefly explain the semantics of the newly introduced syntax.

Pointer variables can be declared like normal variables by using an additional '*', similar to C (pointer variables are initialised as null-pointers). These pointer variables are only allowed to refer to heap data (data created by a new statement), and cannot point to the same data instances as normal variables. The reset statement resets a pointer variable to a null-pointer. The assignment and comparison operators work similar to how they do in C (i.e. whether an assignment is an assignment by reference or by value can depend on the variable declarations). Note that if a data instance allocated by a new statement is no longer reachable in the state graph, then it is destructed. Therefore one could say we employ some form of garbage collection, although this is implicit as non-reachable instances in the state graph are simply not encoded.

It turned out to be relatively easy to combine the two layers of the concrete architecture to construct a model checker for PROM^{+}. In the previous section we mentioned that all that is needed for SoftwareModels was an initial state and all the typing infor-

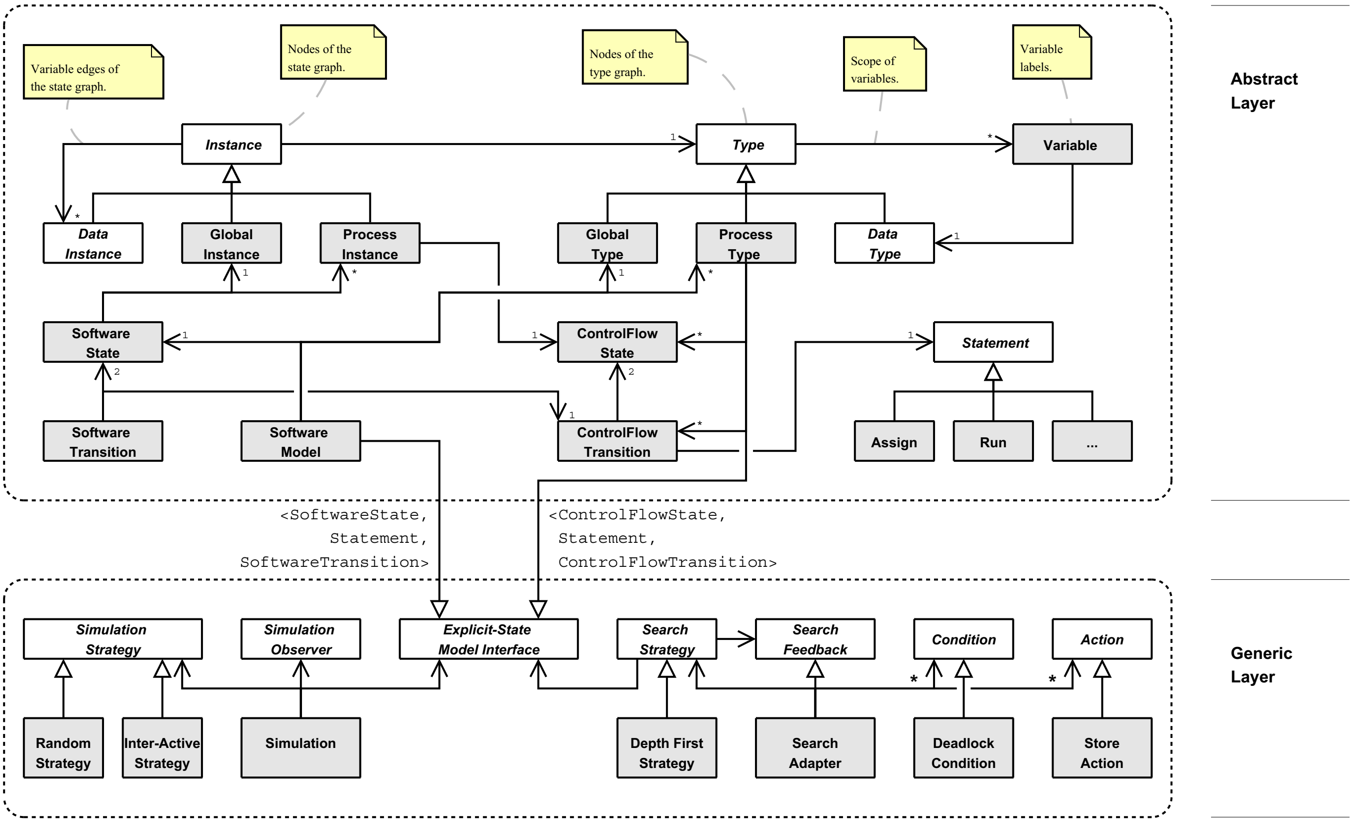


Figure 8 – The conceptual architecture of the framework, divided into a *abstract layer* and an *abstract layer*.

mation. By means of a parser we can generate this information in a straight-forward manner. Once the `SoftwareModel` is constructed, one only has to instantiate the desired algorithms in the generic layer with the appropriate types in order to use them.

An example of a PROM$^+$ model can be found in Listing 5. This is a model specification of the mutual exclusion algorithm by Peterson (as described in [2]). This particular model enables the exploitation of thread-symmetry as the parametrisation of the processes with pointer variables creates state graphs that are thread-symmetrical. In [16] we have developed several models in both PROM$^+$ and equivalent models in a subset of both PROM$^+$ and PROMELA such that we could analyse the effectiveness of our thread-symmetry reduction and to compare the performance of our prototype tool to SPIN.

3.5 RESULTS

The primary new concept of this work is the use of a *layered architecture* in combination with *type parametrisation* to provide reusable algorithms for explicit-state verification. We argue that most of the functionality of our prototype implementation is indeed reusable, and therefore the conceptual architecture does enable reuse in the way we have intended. Not only can reuse be achieved by using the same algorithms for different intermediate representations, different tools could also use the same intermediate representation. For instance, if we have a testing tool and a verification tool for PAs then it makes sense to use the same intermediate representation. Sharing an intermediate representation would improve the interoperability of tools.

The preliminary experiments with the prototype (see [16]) have shown that, with respect to memory consumption (the average size of the bit sequences that represent states), the prototype is comparable or at times even more efficient than SPIN. With respect to time, however, SPIN is still three orders of magnitude faster. Obviously, the design philosophies behind SPIN are directly opposite to those of ours, and it is therefore not surprising that the performance of our tool is worse. SPIN has been continuously optimised to verify PROMELA models as efficiently as possible, thereby making it very difficult to reuse SPIN. In contrast, we sacrifice performance in order to enable reuse. Despite this difference, it is our expectation that we can improve the prototype implementation to achieve performance nearer one order of magnitute slower than SPIN, without sacrificing the principles of our conceptual architecture.

There are a few design choices that have seriously impacted the performance of our tool. Firstly, the choice to use reference counting pointers has placed a significant overhead on everything in the framework. Secondly, the choice to implement reachability algorithms in a modular fashion comes at the cost of a lot of overhead in the form of function calls, which could be avoided by means of more specialised algorithms. Finally, the choice to use graphs to represent states in the intermediate representation comes at the cost of expensive graph operations (such as linearisation). These issues could be improved without changing the principles of our conceptual architecture.

4 FUTURE WORK

The proof-of-concept framework already shows significant potential, but to meet our objectives the framework should be extended in several directions. New generic layers (e.g. for symbolic or bounded model checking) are anticipated. Different intermediate representations (other than the current graph implementation) should be developed for the current explicit-state model interface. Additionally, the functionality in the generic layer should be extended by adding new reusable algorithms. This could include new search strategies or verification of liveness properties (informally addressed in [16]).

A more basic continuation of the framework would be to investigate methods to improve the performance of the framework. This could include reconsidering some design decisions that were made during the creation of the current framework, such as the use of reference counting pointers as well as redesigning the verification functionality in the generic layer.

We expect that the architecture and library will develop into a useful and reusable generic library for formal verification.

REFERENCES

[1] Gerd Behrmann, Alexandre David, and Kim G. Larsen, 'A tutorial on UPPAAL', in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT 2004)*, eds., Marco Bernardo and Flavio Corradini, volume 3185 of *LNCS*, pp. 200–236. Springer–Verlag, (2004).

[2] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1990.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petruccie, Ph. Schnoebelen, and P. McKenzie, *Systems and Software Verification: Model-checking techniques and tools*, Springer-Verlag, 2001.

[4] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis, 'The IF TOOLSET', in *International School on Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, eds., Marco Bernardo and Flavio Corradini, volume 3185 of *LNCS*, pp. 237–267. Springer-Verlag, (2004).

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang, 'Symbolic model checking: 10^{20} states and beyond', in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, ed., John Mitchell, pp. 428–439. IEEE Computer Society Press, (1990).

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, 'Progress on the state explosion problem in model checking', in *Informatics - 10 Years Back. 10 Years Ahead*, ed., Reinhard Wilhelm, volume 2000 of *LNCS*, pp. 176–194. Springer-Verlag, (2001).

[7] Edmund M. Jr. Clarke, Orna Grumberg, and Doron A. Peled, *Model Checking*, MIT Press, 1999.

[8] Rance Cleaveland and Steve Sims, 'The NCSU Concurrency Workbench', in *8th International Conference on Computer Aided Verification (CAV 1996)*, eds., Rajeev Alur and Thomas A. Henzinger, volume 1102 of *LNCS*, pp. 394–397. Springer-Verlag, (1996).

[9] T. Courtney, D. Daly, S. Derisavi, V. Lam, and W. H. Sanders, 'The MÖBIUS modeling environment', in *Tools of the 2003 Illinois International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems*, Research report no. 781/2003, pp. 34–37. Universität Dortmund Fachbereich Informatik, (2003).

[10] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster, 'The MÖBIUS framework and its implementation', *IEEE Trans. Softw. Eng.*, **28**(10), 956–969, (2002).

[11] Salem Derisavi, Peter Kemper, William H. Sanders, and Tod Courtney, 'The MÖBIUS state-level abstract functional interface', *Perform. Eval.*, **54**(2), 105–128, (2003).

[12] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby, 'Building your own software model checker using the BOGOR extensible model checking framework', in *17th International Conference on Computer Aided Verification (CAV 2005)*, eds., Kousha Etessami and Sriram K. Rajamani, volume 3576 of *LNCS*, pp. 148–152. Springer-Verlag, (2005).

[13] Gerard J. Holzmann, *The* SPIN *Model Checker – Primer and Reference Manual*, Addison-Wesley, 2004.

[14] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis, 'On nested depth-first search', in *The Spin Verification System*, eds., Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, (1996).

[15] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli, 'Multi-valued decision diagrams: theory and applications', *International Journal on Multiple-Valued Logic*, **4**(1-2), 9–62, (1998).

[16] Mark Kattenbelt, *Towards an explicit-state model checking framework*, Master's thesis, University of Twente, Enschede, The Netherlands, 2006. (available from `http://www.cs.bham.ac.uk/~mxk`).

[17] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[18] Arend Rensink, 'Towards model checking graph grammars', in *Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, eds., Michael Leuschel, Stefan Gruner, and Stphane Lo Presti, Technical Report DSSE–TR–2003–2, pp. 150–160. University of Southampton, (2003).

[19] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif, 'Space-reduction strategies for model checking dynamic software', *Electronic Notes in Theoretical Computer Science*, **89**(3), (2003).

[20] Claus Schröter, Stefan Schwoon, and Javier Esparza, 'The Model-Checking Kit', in *24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, eds., Wil M. P. van der Aalst and Eike Best, volume 2679 of *LNCS*, pp. 463–472. Springer-Verlag, (2003).

[21] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, third edn., 2000.

Eric Verhulst, Gjalt de Jong (Open License Society, Leuven, Belgium)
E-mail: eric.verhulst@OpenLicenseSociety.org, gjalt.dejong@OpenLicenseSociety.org,

Lessons from developing the OpenComRTOS distributed Real Time Operating System using formal modeling techniques

Abstract

OpenComRTOS is one of the first Real-Time Operating Systems (RTOS) developed using formal modeling techniques. The goal of this project was to obtain a proven trustworthy component as well as a clean and high performance architecture. These goals were achieved. The result is a scalable network-centric operating system with real-time capabilities. In the coarse of the project several lessons were obtained on the use of formal techniques. While the formal modeling resulted in important algorithmic innovations in the RTOS domain for better safety and real-time properties, it was also found that formal modeling has limitations and that the process of peer review is very important in achieving good results. Areas of research in formal model checkers were found as well. In the project we used the TLA/TLC formal modeling toolset of Leslie Lamport.

Systems Engineering approach

The Systems Engineering approach adopted by Open License Society is a classical one as defined in [3] but adapted to the needs of embedded software development. It is first of all an evolutionary process, basically a V-method process, but using constant iteration reviews. In such a process much attention is paid to an incremental development requiring often review meetings by several of the stakeholders. In the case of OpenComRTOS, this started by elaborating a first set of requirements and specifications. Next an initial architecture was defined. Starting from this point two groups started to work in parallel. The first group worked out an architectural model while a second group (led by Prof. Boute of the University in Gent) developed an initial formal model using TLA+/TLC [2]. This model was incrementally refined. At each review meeting between the software engineers developing the architecture and the formal modeling engineer, more details were added to the model, the model was checked for correctness and a new iteration started. This process was stopped when the formal model was deemed close enough to the implementation architecture. Next, a simulation model was developed on a PC (using Windows NT as virtual target). This code was then ported to a real 16bit microcontroller of Melexis [5] and optimized. The software was written in ANSI C and verified with a MISRA rule checker.

Lessons from using formal modeling

The initial goal of using formal techniques was to proof that the developed software was correct. This is an often heard statement from the formal techniques community. A first surprise was that each model gave no errors when verified by the TLC model checker. This is actually due to the iterative nature of the model development process and partly its strength. From an initial rather abstract model successive models are developed by checking them using the model checker and hence each model is correct when the model checker finds no illegal states. As such model checkers can't proof that the software is correct. They can only proof that the formal model is correct.

Other issues were discovered in relation to the use of formal modeling. A first issue is that the TLC model checker declares every action as a critical section, whereas e.g. in the case of a RTOS, many components operate concurrently and real-time performance dictates that on a real target the critical sections are kept as short as possible. While this dictates the avoidance of shared datastructures, it would be helpful to have formal modelers that indicate the real critical sections.

The final issue is the well known problem of state space explosion. Just modeling a small OpenComRTOS application the TLC model checkers has to examine a few million states, exponentially taking more time for every task added to the model. This also requires increasing amounts of memory.

As was outlined above, the use of formal modeling was found to result in a much better architecture. This benefit is the result of the process of successive iteration and review, but also because formal models checkers provide a level of abstraction away from the implementation. In the project e.g. we found that the semantics associated with specific terms used when programming involuntarily influence choices made by the architecting engineer. E.g. a waiting list is associated just with waiting but one overlooks that it also provides buffering behavior. Similarly, even if there was a short learning curve to master the mathematical notation in TLA, with hindsight this was an advantage vs. e.g. using SPIN [7] that uses a C-like syntax.

Results obtained on the MLX16 microcontroller

We shortly summarize the results obtained. OpenComRTOS was defined in layers and uses prioritized packet switching. On each processor an instance of the kernel task provides preemptive, priority based scheduling between "Tasks", each having their own context and to provide inter-task synchronization and communication services using intermediate "Port" objects in the tradition of Hoare's CSP channels but allowing multiple waiters and asynchronous communication. The use of packets simplifies the implementation of inter-task and processor communication, memory management and of kernel services. Although fully written in ANSI-C (except e.g. the context switch), the kernel could be reduced to less than 1 Kbytes single processor and 2 Kbytes with multi-processor support. This so-called L0-kernel also features buffer management that is free from the risk of overflows. The second level is the so-called L1 layer. This layer implements the more tradition services like events, semaphores, FIFOs, mailboxes, memory pools and resource management. The formal modeling was instrumental in discovering that all such services can be seen as special cases derived from a more general "Hub" object. Formal modeling also allowed to discover a better way of handling priority based scheduling in the context of resource sharing. This was remarkable as the engineers in the team had 15 years experience developing a commercial RTOS. Most likely, the same improvement could be applied to all RTOS on the market.

All code is written in ANSI-C, checked for satisfying the MISRA-C rules. Nevertheless, the code is very portable and very easy to maintain.

Conclusion

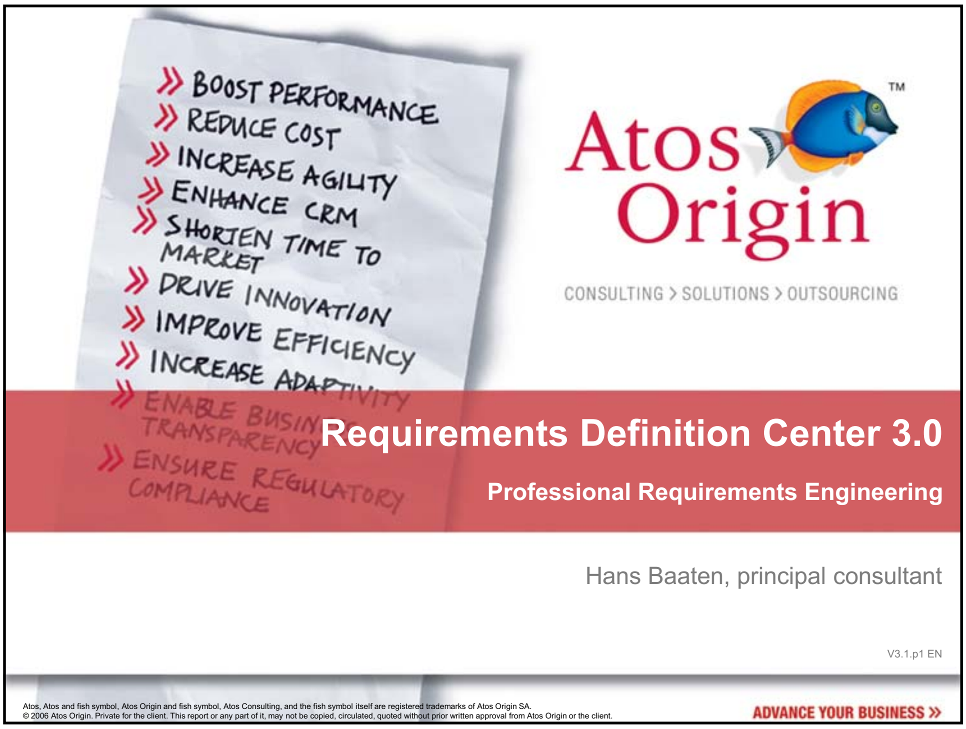
The OpenComRTOS project has shown that even for software domains often associated with 'black art' programming, formal modeling works very well. The resulting software is not only very robust and maintainable but also very performing in size and timings and inherently safer than a standard implementation architecture. It's use however must be integrated with a global systems engineering approach as the process of incremental development and modeling is as important as using the formal model checker itself.

Acknowledgements

The OpenComRTOS project is partly funded under an IWT project for the Flemish Government in Belgium. The formal modeling activities are provided by the University of Gent. Melexis is co-sponsoring the effort by providing the Melexis microcontroller as a resource constrained target for use in embedded automotive electronics.

REFERENCES

1. OpenComRTOS architectural design document on www.OpenLicenseSociety.org
2. TLA+/TLC home page on http://research.microsoft.com/users/lamport/tla/tla.html
3. INCOSE www.incose.org
4. Open License Society www.OpenLicenseSociety.org
5. www.Melexis.com
6. www.spin.org
7. www.misra.org



» BOOST PERFORMANCE
» REDUCE COST
» INCREASE AGILITY
» ENHANCE CRM
» SHORTEN TIME TO MARKET
» DRIVE INNOVATION
» IMPROVE EFFICIENCY
» INCREASE ADAPTIVITY
» ENABLE BUSINESS TRANSPARENCY
» ENSURE REGULATORY COMPLIANCE
Atos Origin
CONSULTING > SOLUTIONS > OUTSOURCING

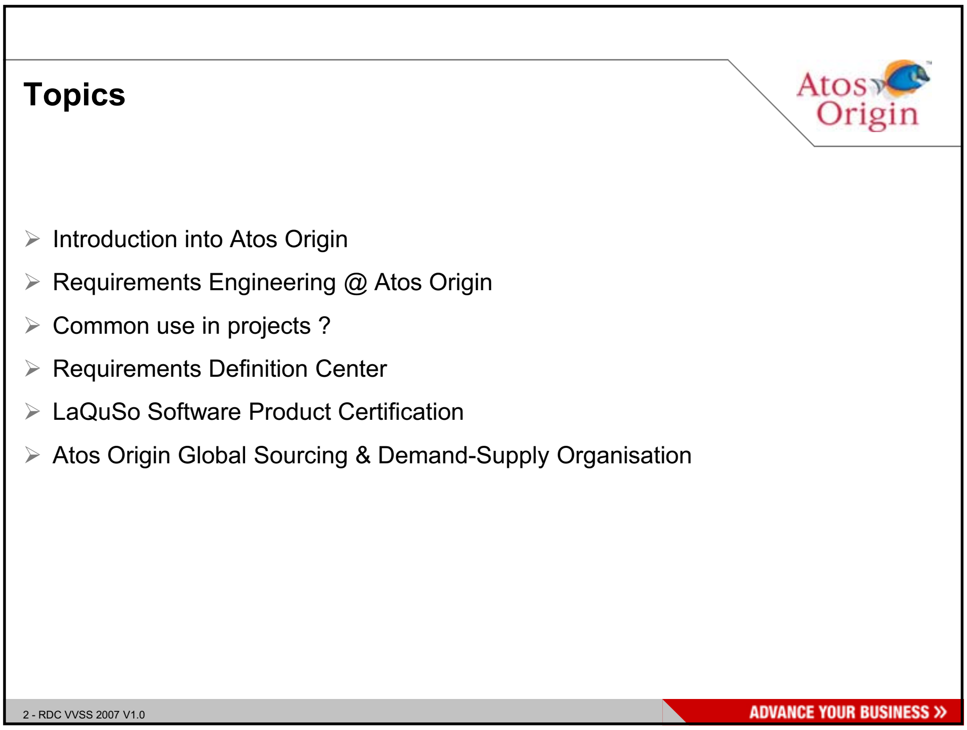
Requirements Definition Center 3.0

Professional Requirements Engineering

Hans Baaten, principal consultant

V3.1.p1 EN

ADVANCE YOUR BUSINESS »



Topics

- Introduction into Atos Origin
- Requirements Engineering @ Atos Origin
- Common use in projects ?
- Requirements Definition Center
- LaQuSo Software Product Certification
- Atos Origin Global Sourcing & Demand-Supply Organisation

2 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

An International Player

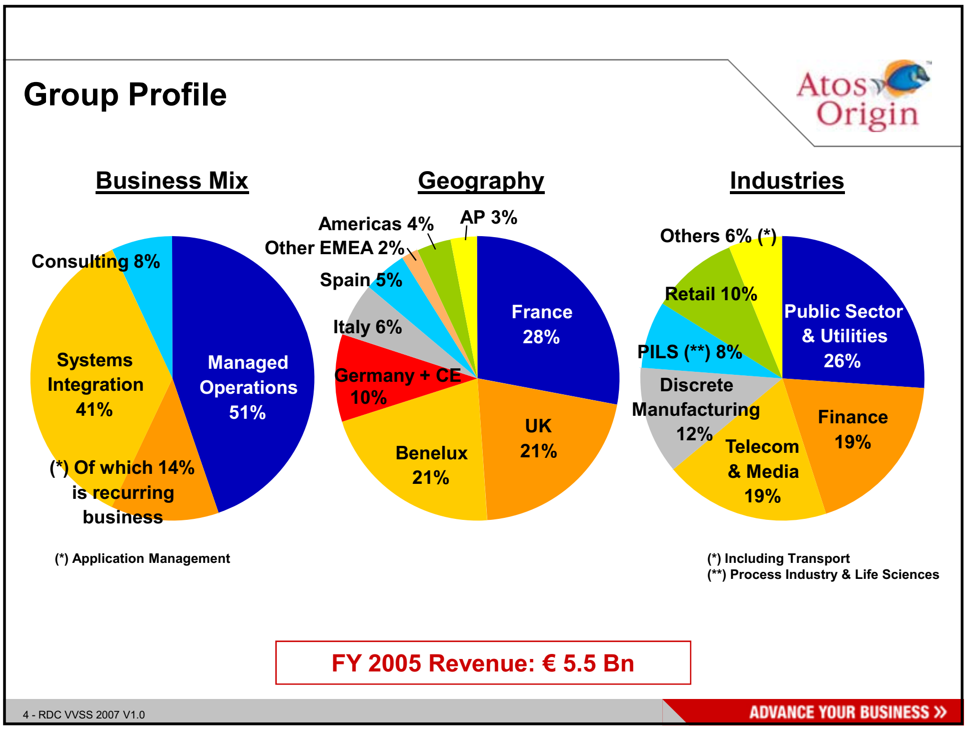
Atos Origin

A leading IT services company providing business consulting, systems integration and managed operations that improve the effectiveness of its clients' Businesses

- Annual revenues of € 5.5 bn
- Over 47,000 employees
- In 40 countries

3 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Group Profile

Atos Origin

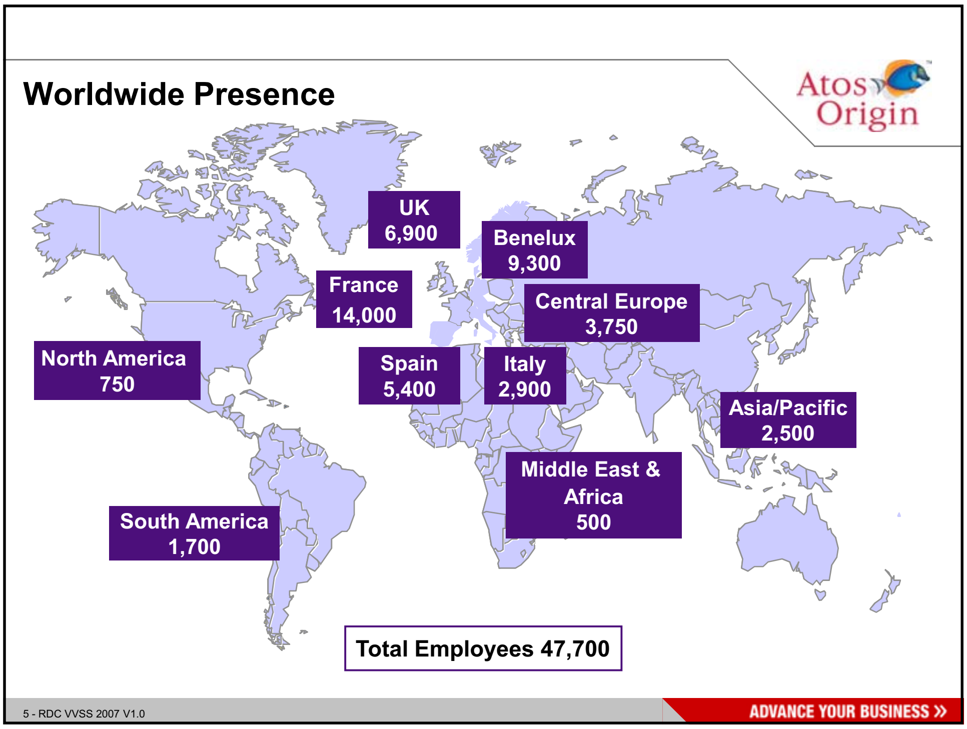
(*) Application Management

(*) Including Transport
(**) Process Industry & Life Sciences

FY 2005 Revenue: € 5.5 Bn

4 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Worldwide Presence

Atos Origin

UK 6,900

Benelux 9,300

France 14,000

Central Europe 3,750

North America 750

Spain 5,400

Italy 2,900

Asia/Pacific 2,500

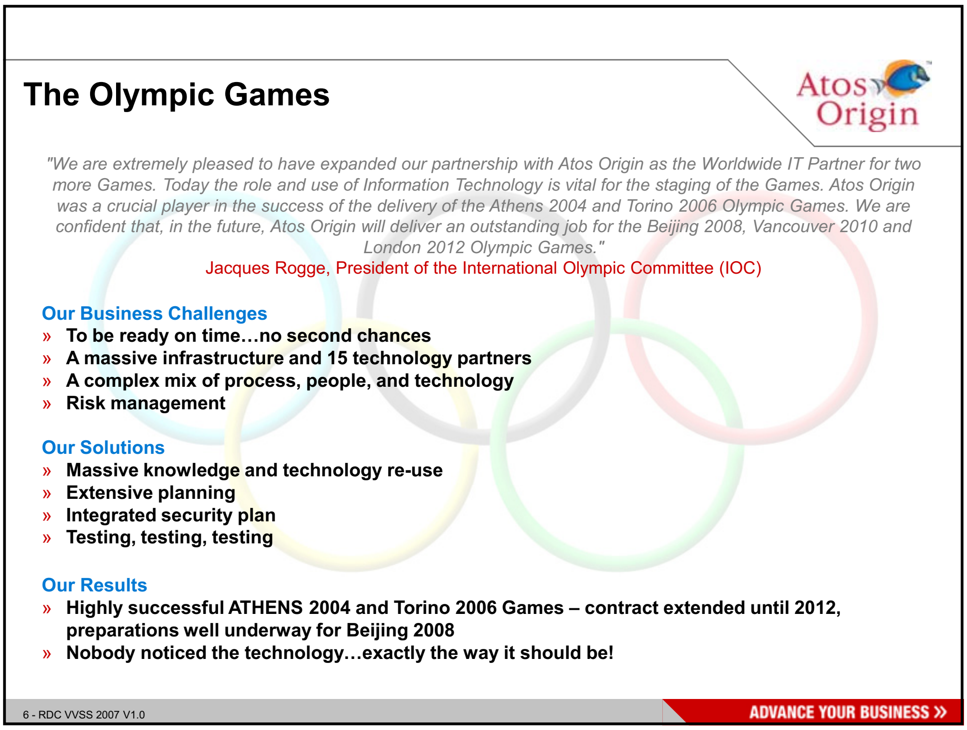
Middle East & Africa 500

South America 1,700

Total Employees 47,700

5 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



The Olympic Games

Atos Origin

"We are extremely pleased to have expanded our partnership with Atos Origin as the Worldwide IT Partner for two more Games. Today the role and use of Information Technology is vital for the staging of the Games. Atos Origin was a crucial player in the success of the delivery of the Athens 2004 and Torino 2006 Olympic Games. We are confident that, in the future, Atos Origin will deliver an outstanding job for the Beijing 2008, Vancouver 2010 and London 2012 Olympic Games."

Jacques Rogge, President of the International Olympic Committee (IOC)

Our Business Challenges

- **To be ready on time…no second chances**
- **A massive infrastructure and 15 technology partners**
- **A complex mix of process, people, and technology**
- **Risk management**

Our Solutions

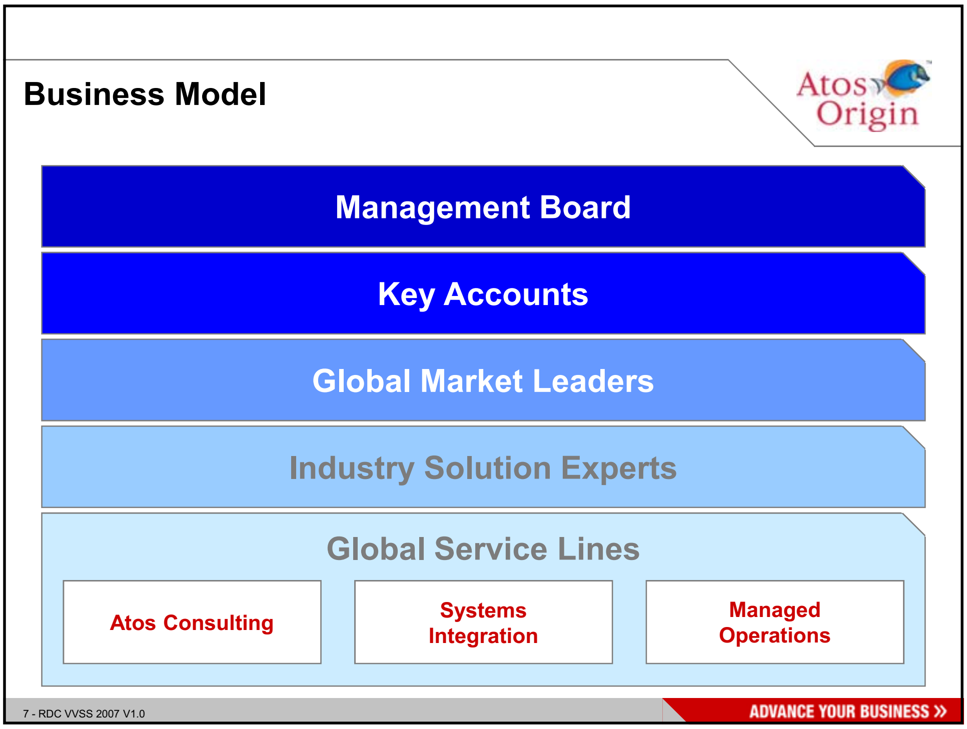
- **Massive knowledge and technology re-use**
- **Extensive planning**
- **Integrated security plan**
- **Testing, testing, testing**

Our Results

- **Highly successful ATHENS 2004 and Torino 2006 Games – contract extended until 2012, preparations well underway for Beijing 2008**
- **Nobody noticed the technology…exactly the way it should be!**

6 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Business Model

Management Board

Key Accounts

Global Market Leaders

Industry Solution Experts

Global Service Lines

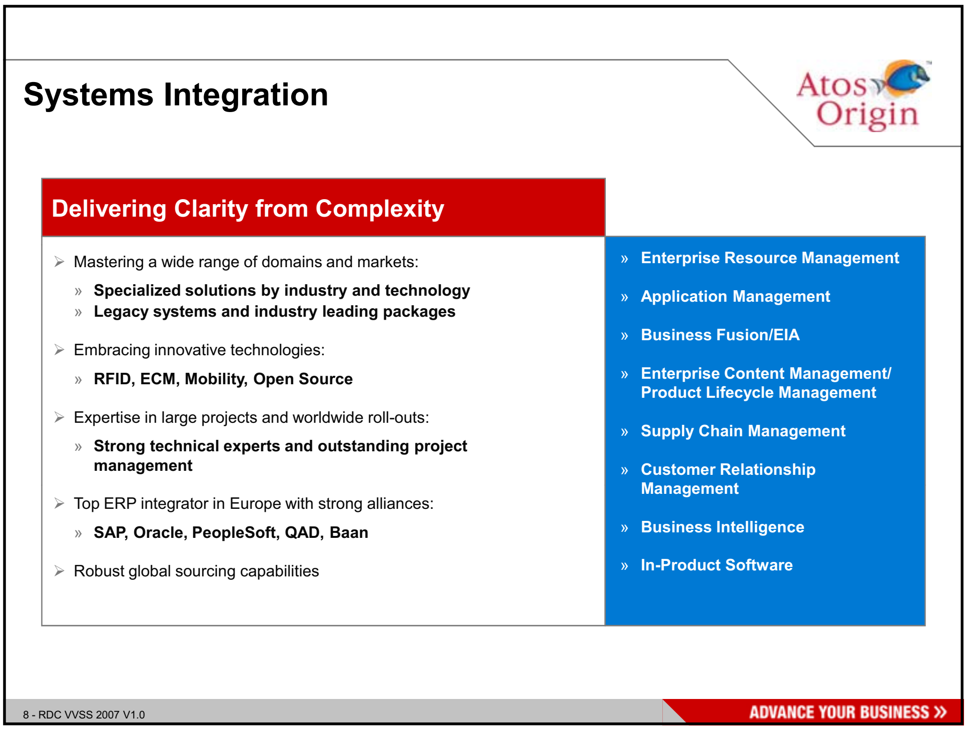
Atos Consulting

Systems Integration

Managed Operations

7 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Systems Integration

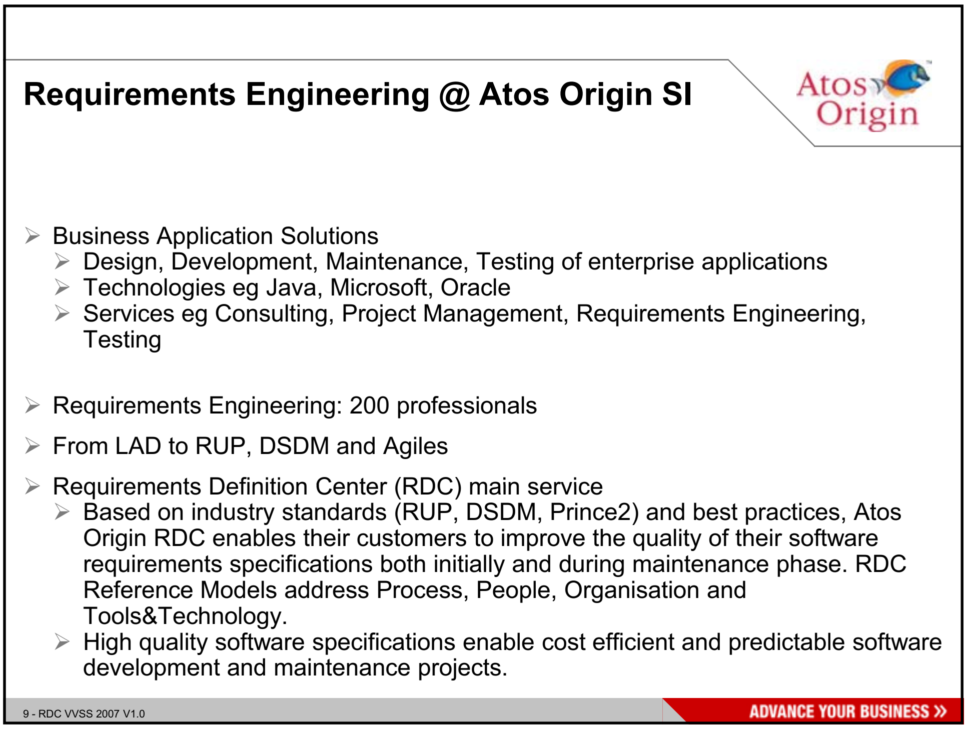
Delivering Clarity from Complexity

- Mastering a wide range of domains and markets:
  - **Specialized solutions by industry and technology**
  - **Legacy systems and industry leading packages**
- Embracing innovative technologies:
  - **RFID, ECM, Mobility, Open Source**
- Expertise in large projects and worldwide roll-outs:
  - **Strong technical experts and outstanding project management**
- Top ERP integrator in Europe with strong alliances:
  - **SAP, Oracle, PeopleSoft, QAD, Baan**
- Robust global sourcing capabilities

- **Enterprise Resource Management**
- **Application Management**
- **Business Fusion/EIA**
- **Enterprise Content Management/ Product Lifecycle Management**
- **Supply Chain Management**
- **Customer Relationship Management**
- **Business Intelligence**
- **In-Product Software**

8 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

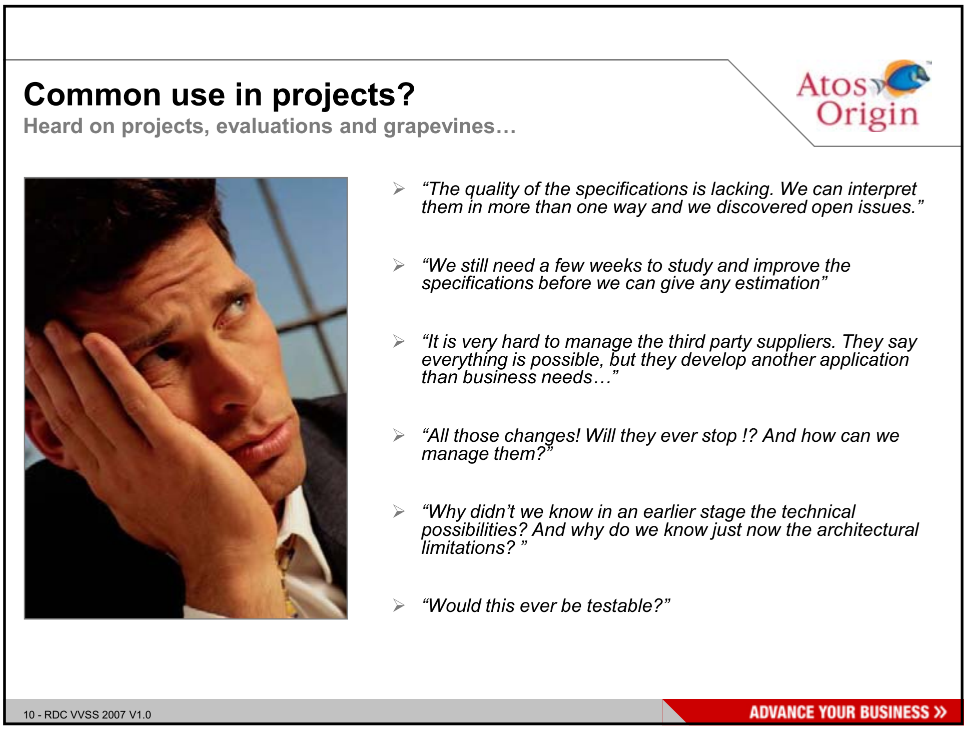


Requirements Engineering @ Atos Origin SI

- Business Application Solutions
  - Design, Development, Maintenance, Testing of enterprise applications
  - Technologies eg Java, Microsoft, Oracle
  - Services eg Consulting, Project Management, Requirements Engineering, Testing
- Requirements Engineering: 200 professionals
- From LAD to RUP, DSDM and Agiles
- Requirements Definition Center (RDC) main service
  - Based on industry standards (RUP, DSDM, Prince2) and best practices, Atos Origin RDC enables their customers to improve the quality of their software requirements specifications both initially and during maintenance phase. RDC Reference Models address Process, People, Organisation and Tools&Technology.
  - High quality software specifications enable cost efficient and predictable software development and maintenance projects.

9 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



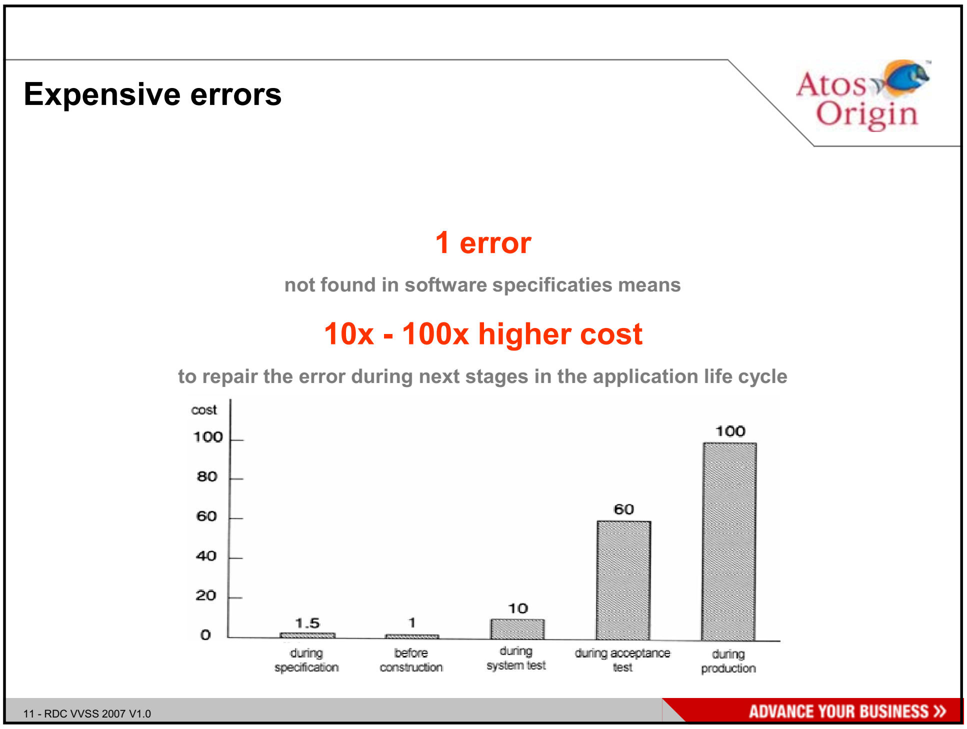
Common use in projects?

Heard on projects, evaluations and grapevines…

- *“The quality of the specifications is lacking. We can interpret them in more than one way and we discovered open issues.”*
- *“We still need a few weeks to study and improve the specifications before we can give any estimation”*
- *“It is very hard to manage the third party suppliers. They say everything is possible, but they develop another application than business needs…”*
- *“All those changes! Will they ever stop !? And how can we manage them?”*
- *“Why didn’t we know in an earlier stage the technical possibilities? And why do we know just now the architectural limitations? ”*
- *“Would this ever be testable?”*

10 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Expensive errors

Atos Origin

1 error

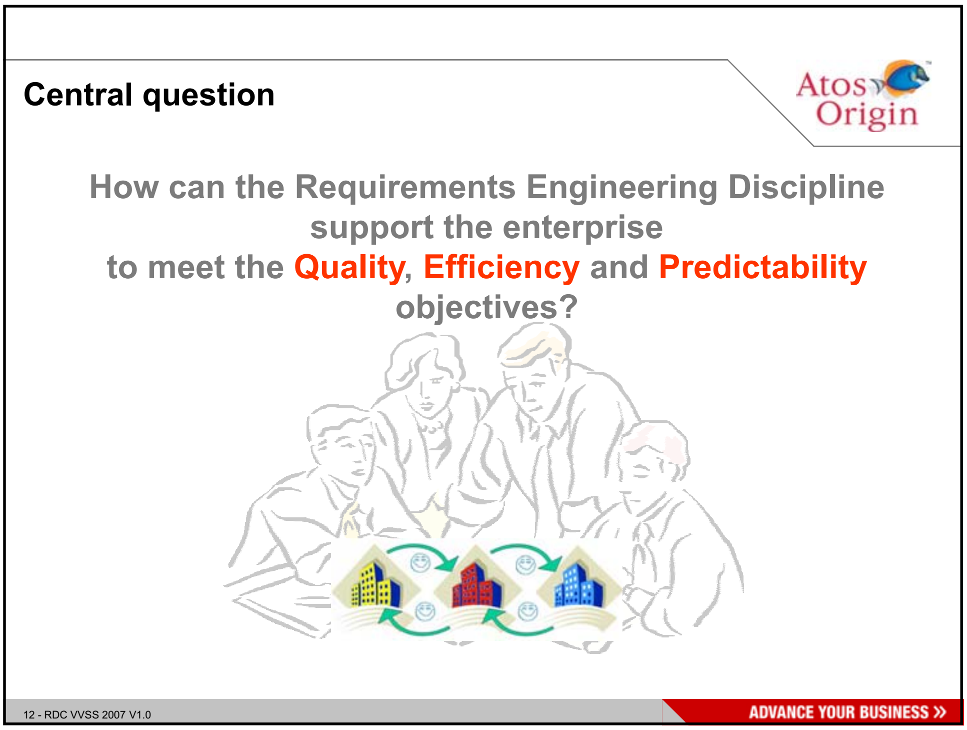
not found in software specificaties means

10x - 100x higher cost

to repair the error during next stages in the application life cycle

11 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



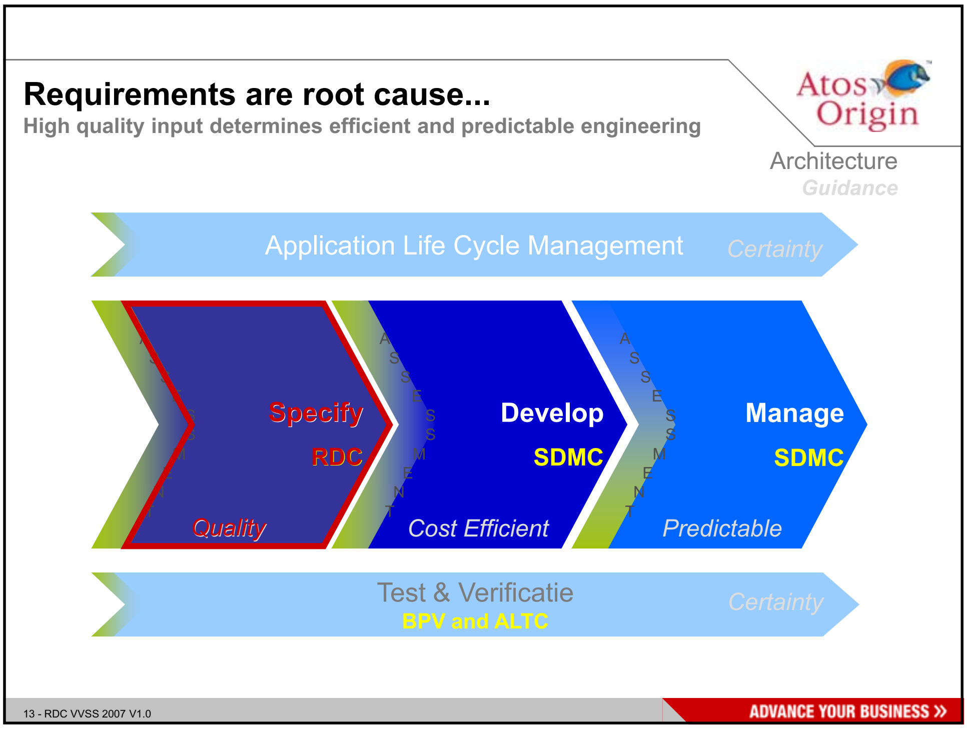
Central question

Atos Origin

How can the Requirements Engineering Discipline support the enterprise to meet the Quality, Efficiency and Predictability objectives?

12 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



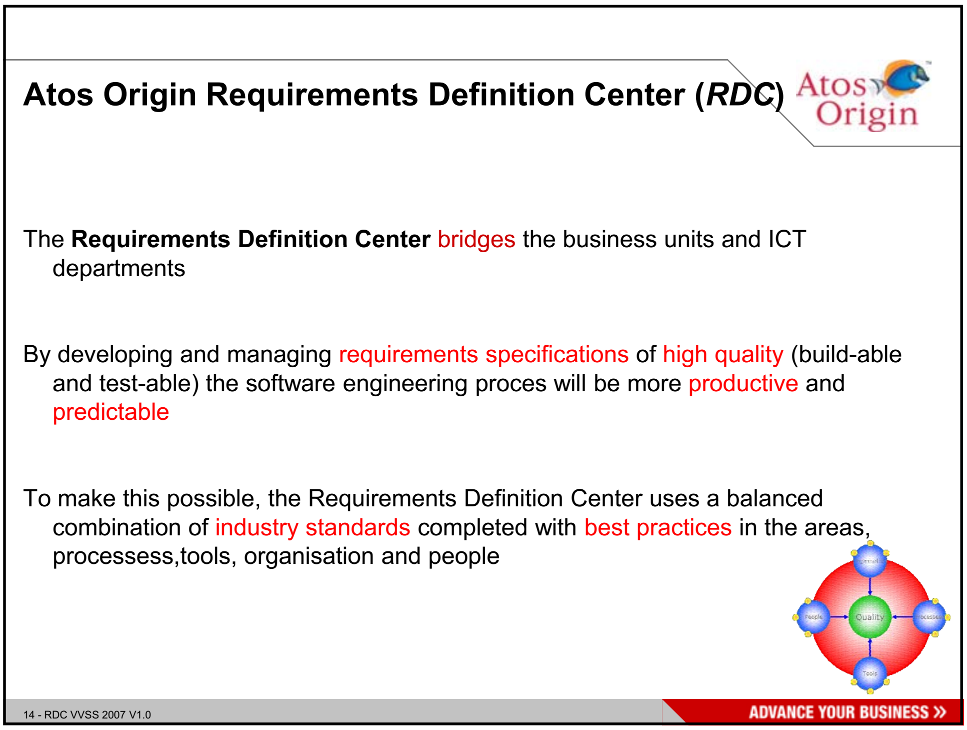
Requirements are root cause...

High quality input determines efficient and predictable engineering

Architecture
Guidance

13 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Atos Origin Requirements Definition Center (*RDC*)

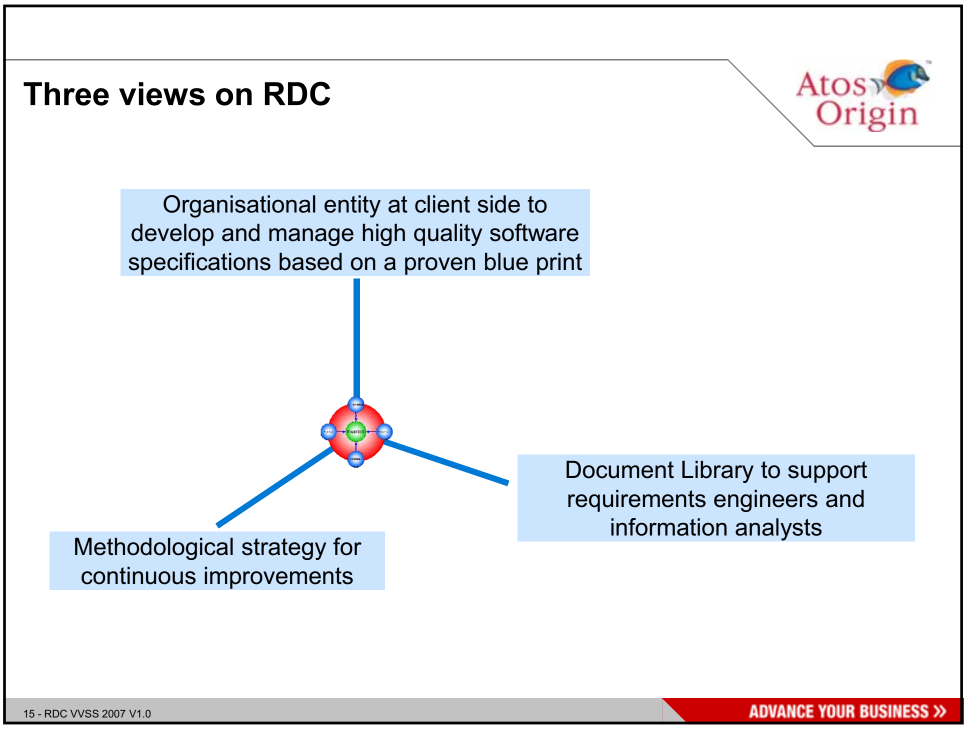
The **Requirements Definition Center** bridges the business units and ICT departments

By developing and managing requirements specifications of high quality (build-able and test-able) the software engineering proces will be more productive and predictable

To make this possible, the Requirements Definition Center uses a balanced combination of industry standards completed with best practices in the areas, processess,tools, organisation and people

14 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Three views on RDC

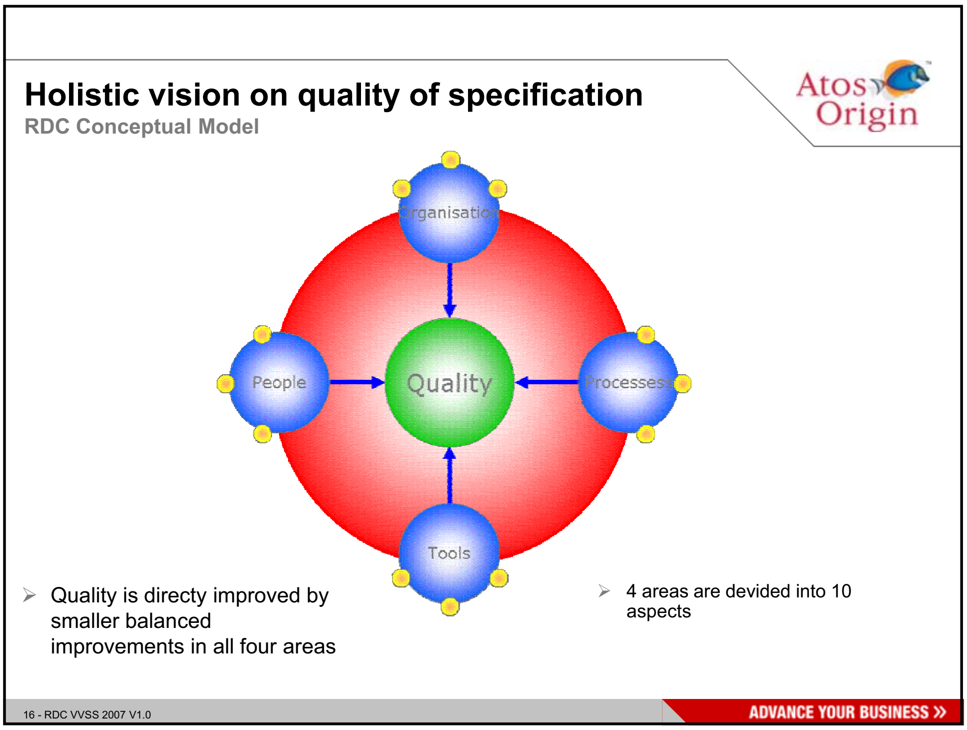
Organisational entity at client side to develop and manage high quality software specifications based on a proven blue print

Document Library to support requirements engineers and information analysts

Methodological strategy for continuous improvements

15 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Holistic vision on quality of specification

RDC Conceptual Model

- Quality is directy improved by smaller balanced improvements in all four areas

- 4 areas are devided into 10 aspects

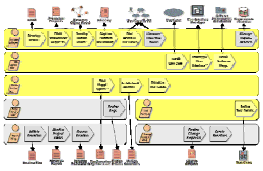
16 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

Processess (1)

Primairy proces Requirements Engineering

- Our *best practices* with RUP
  - Iterative development
  - Use Case driven (specification and testing)
  - Requirements management
  - Continuous quality verification
  - Change Control Management

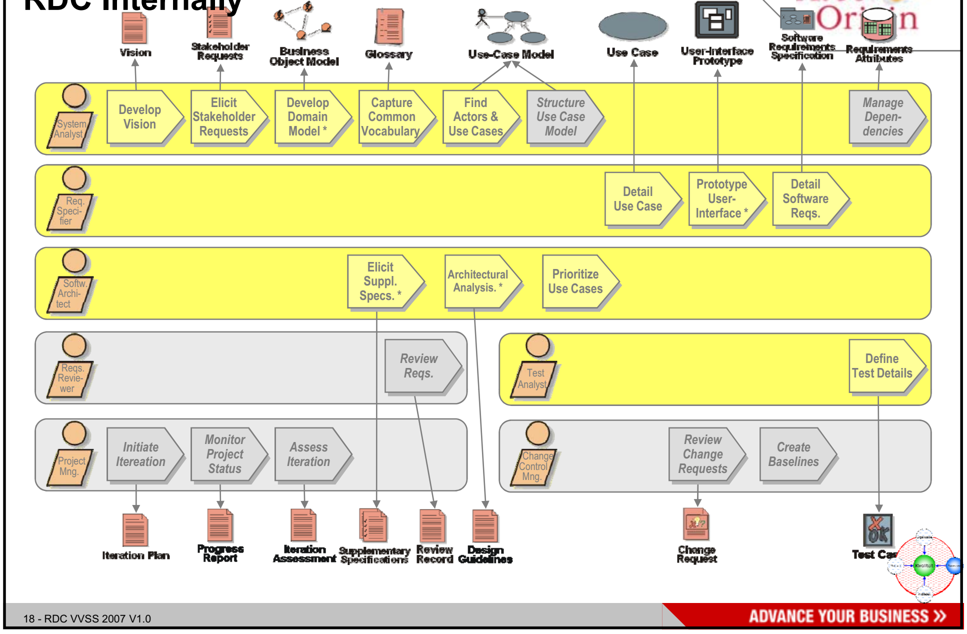


- Contemporary elicitation techniques
  - white board & brown paper sessions
  - use case sessions
  - workshops

17 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

RDC Internally



Vision

Stakeholder Requests

Business Object Model

Glossary

Use-Case Model

Use Case

User-Interface Prototype

Software Requirements Specification

Requirements Attributes

System Analyst: Develop Vision; Elicit Stakeholder Requests; Develop Domain Model *; Capture Common Vocabulary; Find Actors & Use Cases; Structure Use Case Model; Manage Depen-dencies

Req. Speci-fier: Detail Use Case; Prototype User-Interface *; Detail Software Reqs.

Softw. Archi-tect: Elicit Suppl. Specs. *; Architectural Analysis. *; Prioritize Use Cases

Reqs. Revie-wer: Review Reqs.

Test Analyst: Define Test Details

Project Mng.: Initiate Iteration; Monitor Project Status; Assess Iteration

Change Control Mng.: Review Change Requests; Create Baselines

Iteration Plan

Progress Report

Iteration Assessment

Supplementary Specifications

Review Record

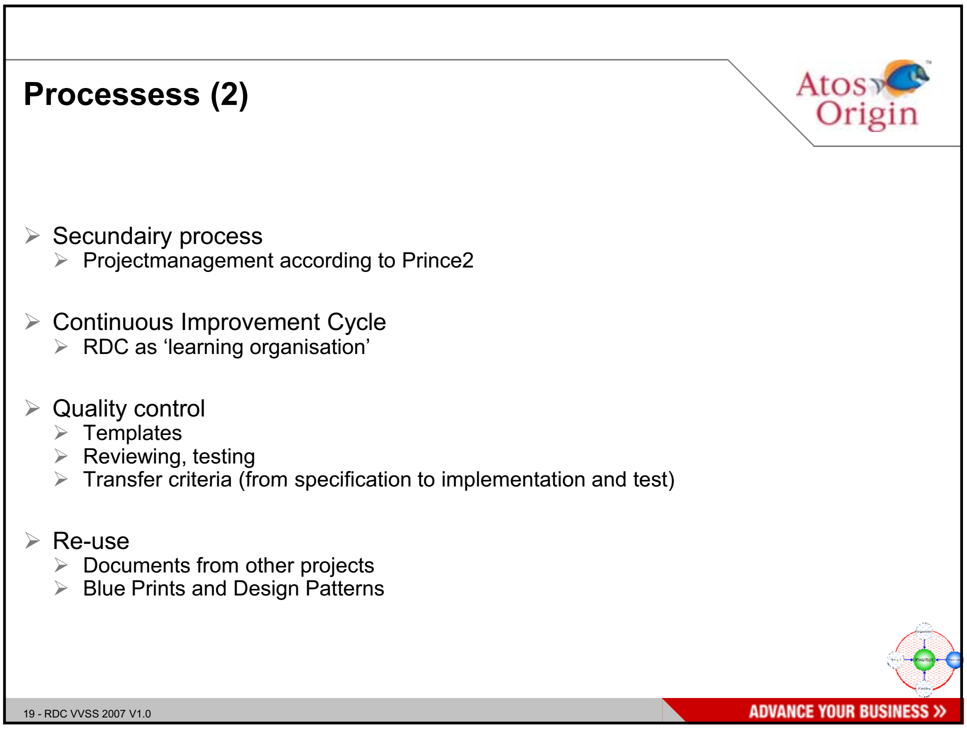
Design Guidelines

Change Request

Test Cas

18 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

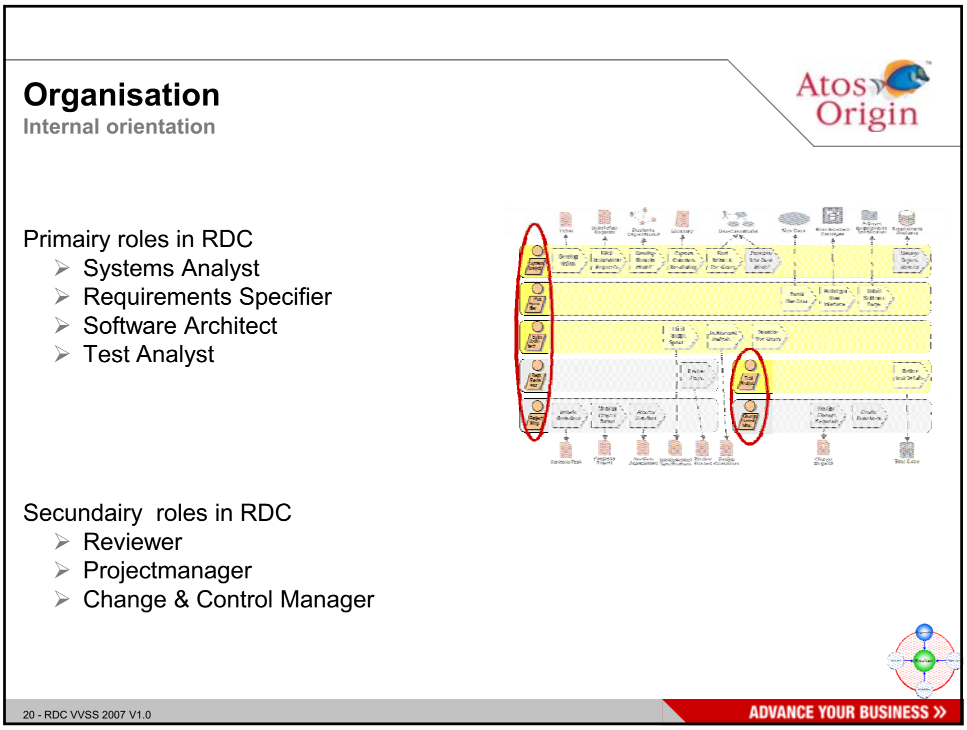


Processess (2)

- Secundairy process
  - Projectmanagement according to Prince2
- Continuous Improvement Cycle
  - RDC as ‘learning organisation’
- Quality control
  - Templates
  - Reviewing, testing
  - Transfer criteria (from specification to implementation and test)
- Re-use
  - Documents from other projects
  - Blue Prints and Design Patterns

19 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Organisation

Internal orientation

Primairy roles in RDC

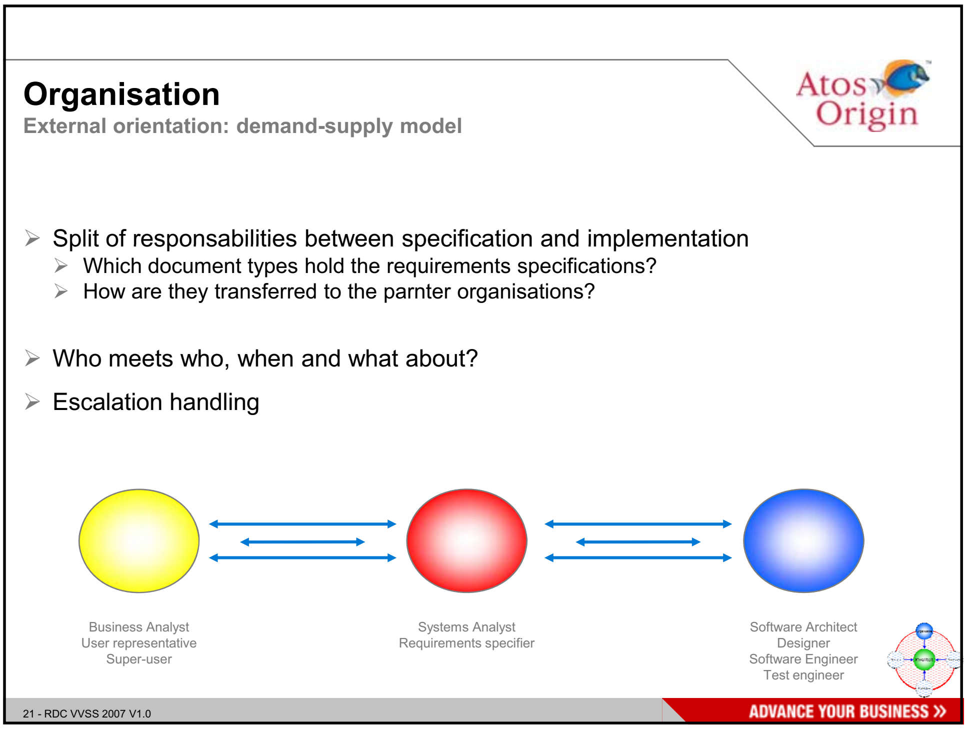
- Systems Analyst
- Requirements Specifier
- Software Architect
- Test Analyst

Secundairy roles in RDC

- Reviewer
- Projectmanager
- Change & Control Manager

20 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Organisation

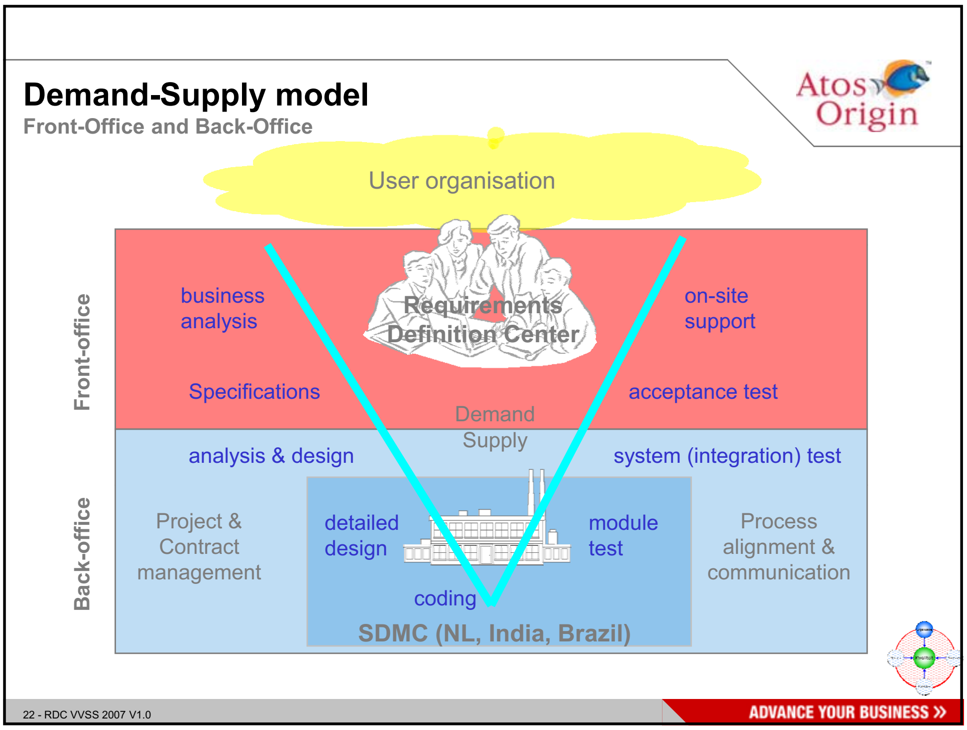
External orientation: demand-supply model

- Split of responsabilities between specification and implementation
  - Which document types hold the requirements specifications?
  - How are they transferred to the parnter organisations?
- Who meets who, when and what about?
- Escalation handling

Business Analyst
User representative
Super-user
Systems Analyst
Requirements specifier
Software Architect
Designer
Software Engineer
Test engineer

21 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Demand-Supply model

Front-Office and Back-Office

22 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

Tooling

Atos Origin

- Infrastructure on (under) your desk
- Toolset
- Working guidelines
- Document Library

Rational software

Borland® Caliber®

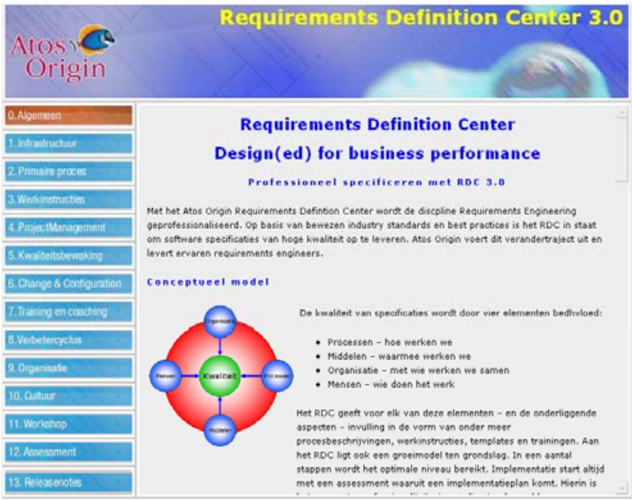
ENTERPRISE ARCHITECT

Compuware Optimal Trace™

ADVANCE YOUR BUSINESS »

Document Library

Atos Origin



Requirements Definition Center 3.0

Requirements Definition Center

Design(ed) for business performance

Professioneel specificeren met RDC 3.0

Met het Atos Origin Requirements Defintion Center wordt de discpline Requirements Engineering geprofessionaliseerd. Op basis van bewezen industry standards en best practices is het RDC in staat om software specificaties van hoge kwaliteit op te leveren. Atos Origin voert dit verandertraject uit en levert ervaren requirements engineers.

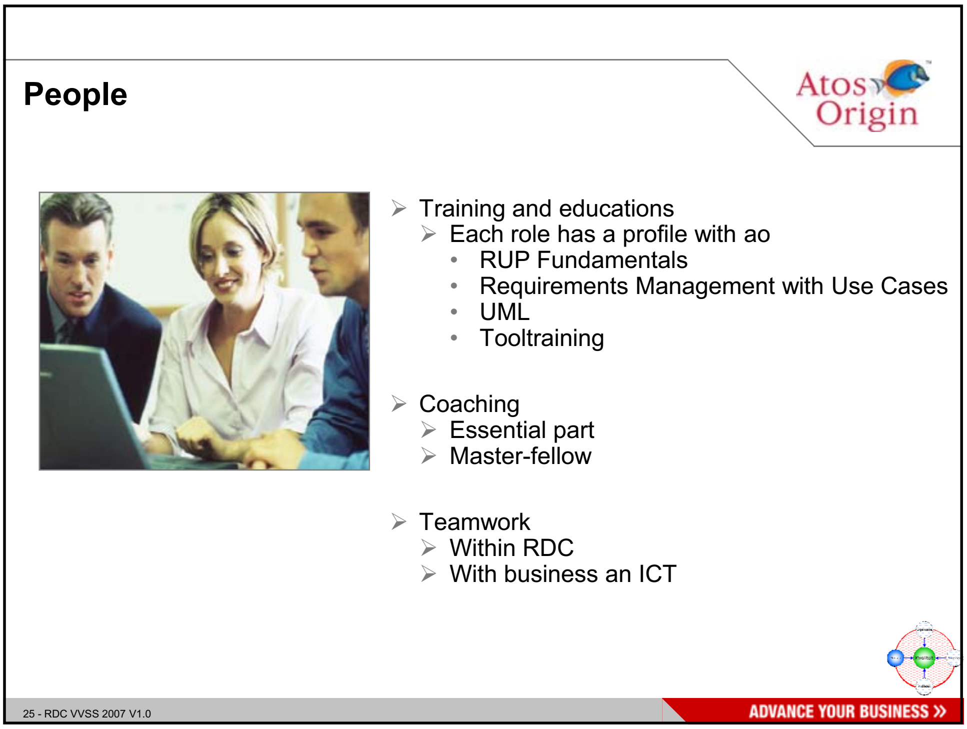
Conceptueel model

De kwaliteit van specificaties wordt door vier elementen bedhvloed:

- Processen – hoe werken we
- Middelen – waarmee werken we
- Organisatie – met wie werken we samen
- Mensen – wie doen het werk

Het RDC geeft voor elk van deze elementen – en de onderliggende aspecten – invulling in de vorm van onder meer procesbeschrijvingen, werkinstructies, templates en trainingen. Aan het RDC ligt ook een groeimodel ten grondslag. In een aantal stappen wordt het optimale niveau bereikt. Implementatie start altijd met een assessment waaruit een implementatieplan komt. Hierin is

ADVANCE YOUR BUSINESS »

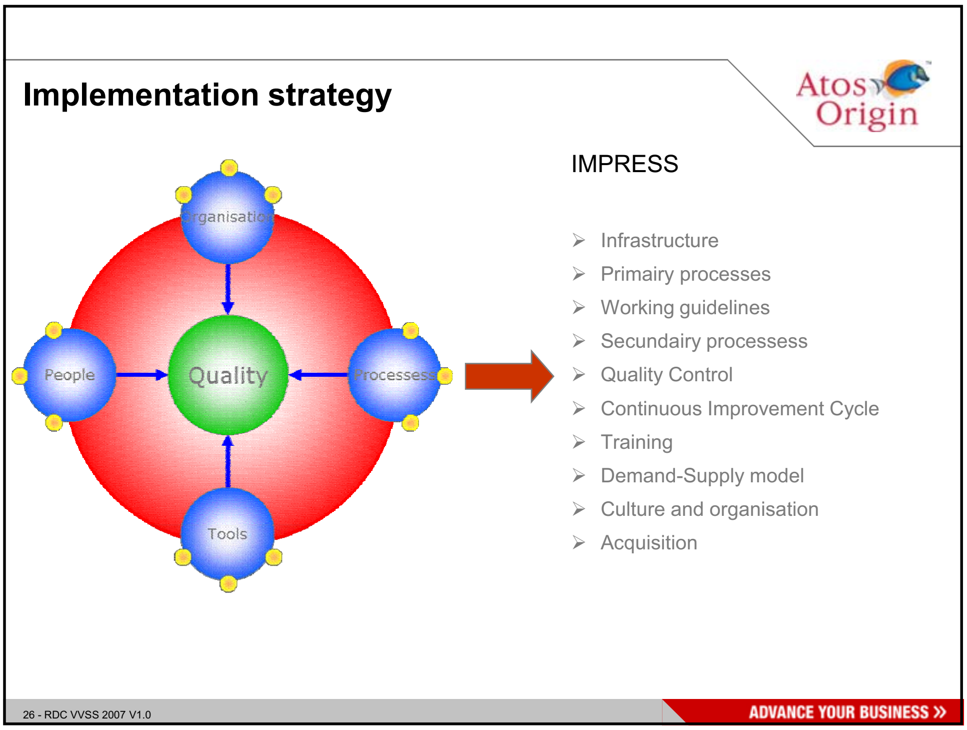


People

- Training and educations
  - Each role has a profile with ao
    - RUP Fundamentals
    - Requirements Management with Use Cases
    - UML
    - Tooltraining
- Coaching
  - Essential part
  - Master-fellow
- Teamwork
  - Within RDC
  - With business an ICT

25 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



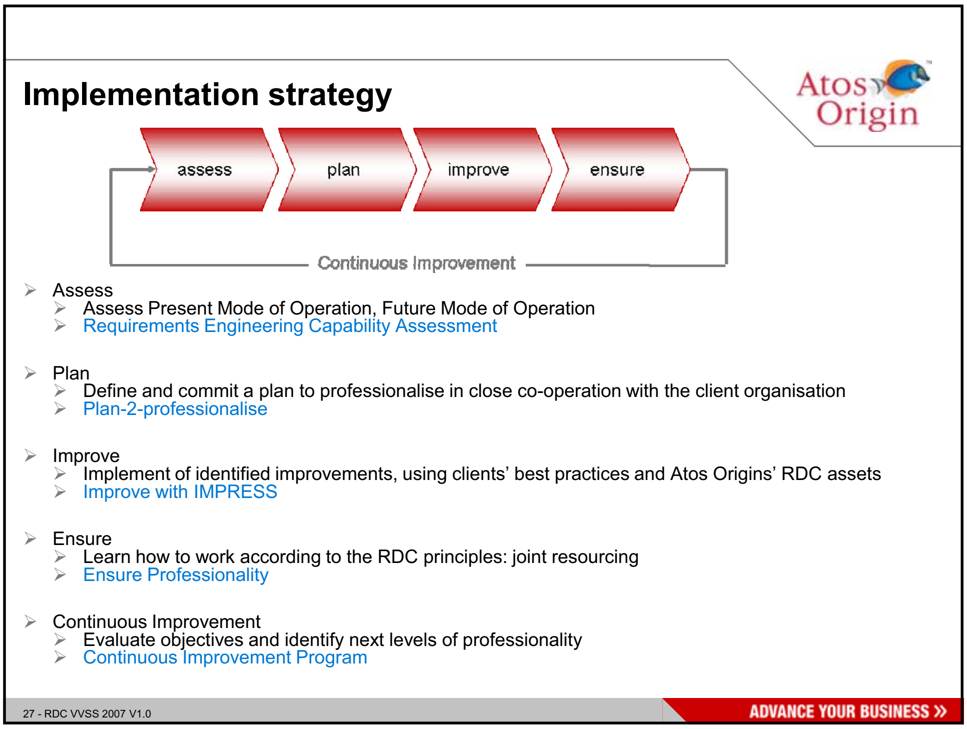
Implementation strategy

IMPRESS

- Infrastructure
- Primairy processes
- Working guidelines
- Secundairy processesss
- Quality Control
- Continuous Improvement Cycle
- Training
- Demand-Supply model
- Culture and organisation
- Acquisition

26 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Implementation strategy

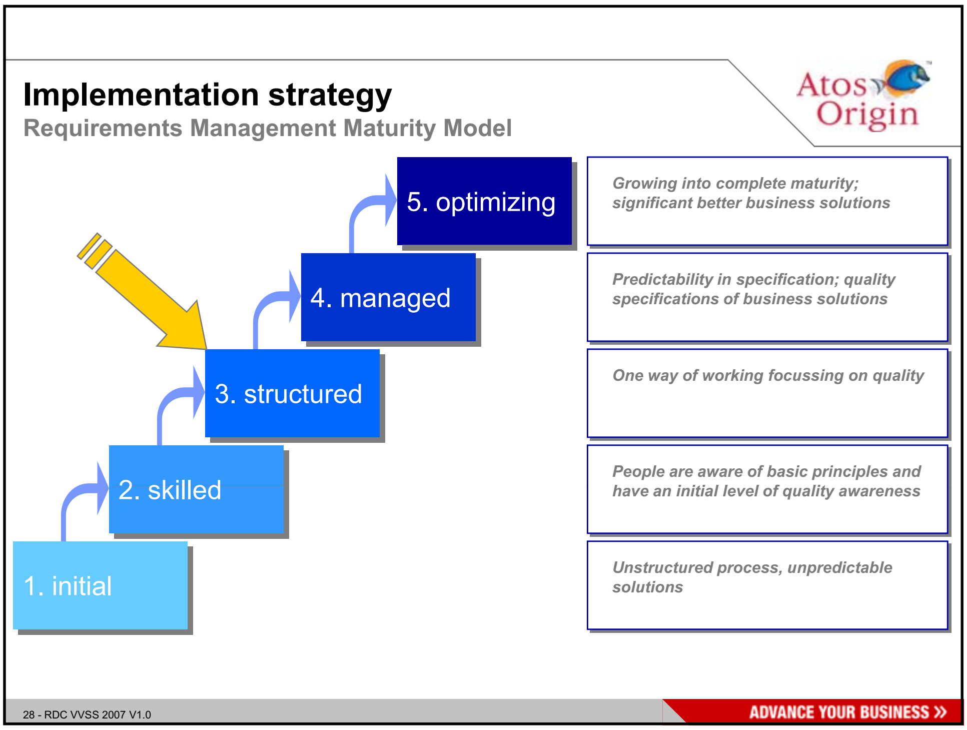
Atos Origin

assess
plan
improve
ensure
Continuous Improvement

- Assess
  - Assess Present Mode of Operation, Future Mode of Operation
  - Requirements Engineering Capability Assessment
- Plan
  - Define and commit a plan to professionalise in close co-operation with the client organisation
  - Plan-2-professionalise
- Improve
  - Implement of identified improvements, using clients' best practices and Atos Origins' RDC assets
  - Improve with IMPRESS
- Ensure
  - Learn how to work according to the RDC principles: joint resourcing
  - Ensure Professionality
- Continuous Improvement
  - Evaluate objectives and identify next levels of professionality
  - Continuous Improvement Program

27 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



Implementation strategy

Requirements Management Maturity Model

Atos Origin

5. optimizing
4. managed
3. structured
2. skilled
1. initial

Growing into complete maturity; significant better business solutions

Predictability in specification; quality specifications of business solutions

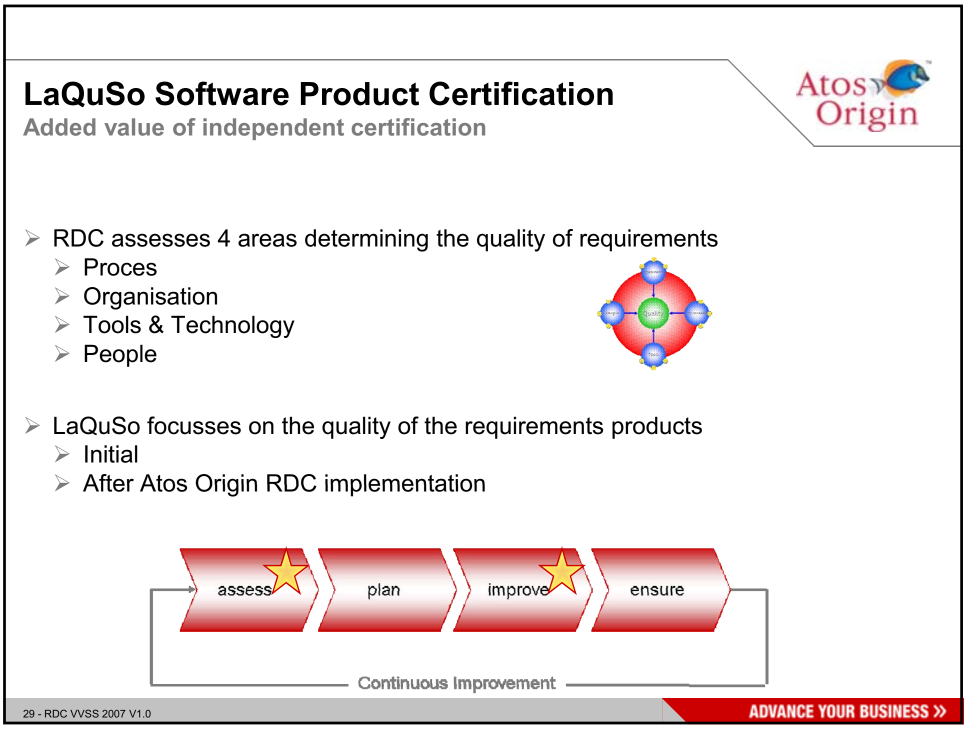
One way of working focussing on quality

People are aware of basic principles and have an initial level of quality awareness

Unstructured process, unpredictable solutions

28 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »



LaQuSo Software Product Certification

Added value of independent certification

- RDC assesses 4 areas determining the quality of requirements
  - Proces
  - Organisation
  - Tools & Technology
  - People
- LaQuSo focusses on the quality of the requirements products
  - Initial
  - After Atos Origin RDC implementation

assess → plan → improve → ensure

Continuous Improvement

29 - RDC VVSS 2007 V1.0

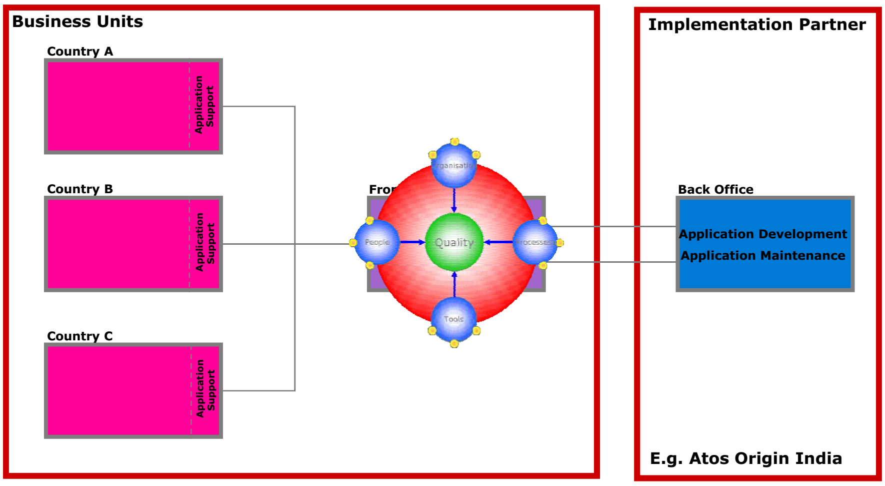
ADVANCE YOUR BUSINESS »

Atos Origin Global Sourcing & Demand-Supply Organisation

30 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

Atos Origin Global Sourcing & Demand-Supply Organisation



31 - RDC VVSS 2007 V1.0

ADVANCE YOUR BUSINESS »

Atos Origin™

CONSULTING › SOLUTIONS › OUTSOURCING

For more information please contact:

Hans Baaten, principal consultant
t +31 (0) 030 – 299 5584
m +31 (0)6 55 122 475
Hans.Baaten@atosorigin.com

www.atosorigin.com

ADVANCE YOUR BUSINESS »

Requirements and qualities

Introduction

There is not one good and clear definition of the term requirement. Every author has virtually his or her own definition. There is however a commonality to be discovered. Requirements are needs that are to be met and which involve client and supplier stakeholders. These needs can be described as functions, qualities and constraints. The requirement type functions describe what the result must do or be able to do, the type qualities describe how well the result must function and the type constraints are solutions that have to be met (or actual constraints that define the border parameters of the requirement itself). From idea to described need, several parts of the organization such as business, IT, are required to be involved before the realisation of the desired solution can begin. Every part of the organization (i.e. all the stakeholders) describes the terms the need in terms of the three types of requirement.

The purposes of requirements are usually more easily agreed upon:

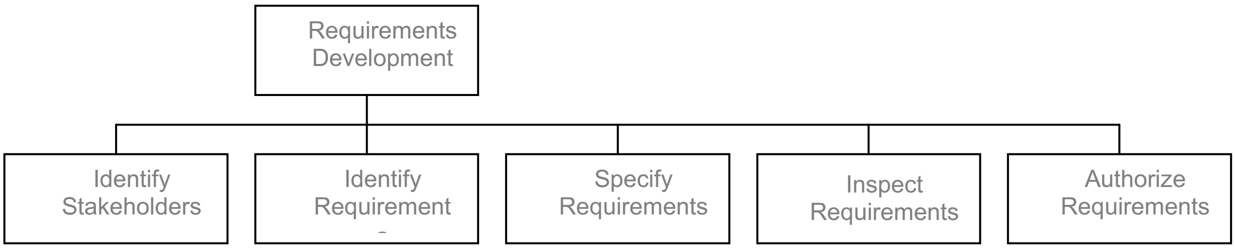
- To match a common picture for the development of products and/or the implementation of processes of organizations;
- To deliver a result which complies with the requirements that have been agreed upon and to enable requirements based testing (RBT);
- Everything that is done within the scope of the project is described in requirements and is laid down in requirements documentation.

Requirements Management

Management of requirements keeps track of the requirements during the development process by means of documentation of requirements. A requirement-tracking table is kept for this purpose often supported by tooling. From this management reports will show the required details for the organization at hand. When changes during the project appear requirements management assures a structured handling conform the requirementsdevelopment process (i.e. Changemanagement).

Requirements Development

Development of requirements is a structured process that generally takes place in five steps:



Every step can be followed as easily by the next step as resulting in going back one or more steps. This process applies to requirement developers, who have at least a general understanding of the described result required. This understanding is necessary to determine whether stakeholders and requirements are identified and selected for the right reasons and understanding the stakeholders during the interviews when specifying the requirements.

Goal

Goals of requirements descibe the needs and expectations of the stakeholder. The needs are the rationale for the products and the desciption of the products in terms of functions. For instance: the time to market has to be shortened. This stakeholder function serves as a rationale for all products in relation for the developtment process. The adequate desciption of products is more easily achieved when put into perspective of this particular goal. It needs no further argumentation that the more adequately (or detailed) products are described, the more succesful the development process shall be.

Goals however can still be difficult to reach. Goals are also difficult to describe; even more difficult to define precisely. When goals are not clear, it is questionable whether products used to reach unclear goals will attribute to the realisation of the goal itself. Why spent resources on the realisation?

Requirements engineering is excellent in determining acurate goals because it emphasizes on describing functions, qualities and constraints.

Value of requirements

Functions are actions or effects that a product or result must do or have. The testing of functions is quite simple. It is enough to determine the existence of the function: is it working or is it not working. With the function alone the goal cannot be achieved, because it constitues no value yet. The value of the function is in its qualities. The quality of a particular function is invariable always measurable. There is always a scale, measure and level involved with qualities. If the function alone exists and it performs below expectations, the value is very low. In this example performance is the scale, the measure could be described in terms of time of money and the level is below expectation. When the level is sufficient, the value rises. When the level is good, the value rises even more. Etc. When there is no qualitylevel at all, the function does not do it's function; is does not exist. A function therefore only exists with its qualities.

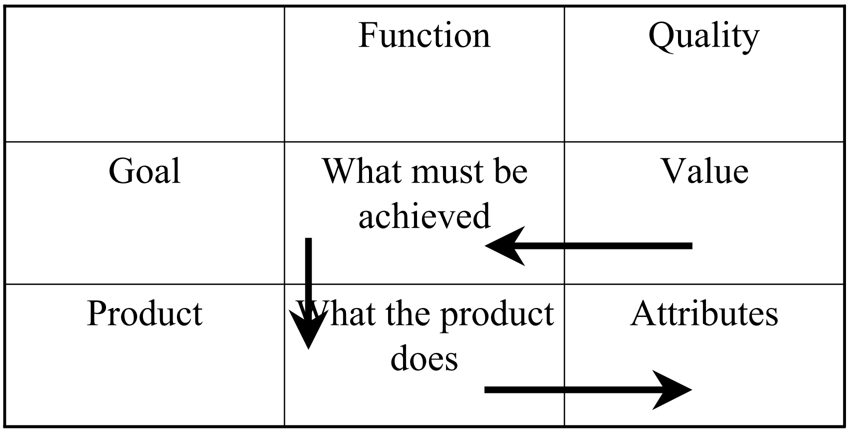
Other literature descibe qualities in terms of non functionals or acception criteria. Non functionals constitute usually of qualities as well as requirements of certain departments such as the legal department. Acception criteria usually become into play during testing. The use of the word "Qualities" is therefore a more appropriate phrase because it indicates what they are: qualities of functions as apposed to non-something or other.

Goal and qualities

As qualities constitue value for functions, they also constitute value for the goals to be reached. In terms of goals, qualities are related to what must be achieved (functions). Goals and related qualities are described in terms of stakeholder language. Defining stakeholder functions and qualities that are unambiguous means they have to be confined within the stakeholder language. This is by no means similar to the more commonly used technical language with product functions and product qualities. Therefore it is imperative that de requirements engineer understands the stakeholder language perfectly well to be able to translate it into technical language.

The following table discribes the most succesful route to develop adequate requirements.

Requirements



	Function	Quality
Goal	What must be achieved	Value
Product	What the product does	Attributes

To explain the significance of this table and the flow indicated by the arrows, the opposite route of the arrows should be explored:

From product function to goal function	Great risk of describing too many product functions that do not contribute to the goal(s). When these functions are developed, one has probably solved someone elses goal at probably the wrong time and incomplete as well.
From goal quality to goal function	Only goal qualities is not enough to succesfully describe requirements. A quality not attached to a function constitutes no value . Qualities are as attached to functions as functions are to qualities.
From goal quality to product quality	This step is to big a step to take because it is not clear to what function(s) they are attached. This results in general qualities that can not be used to define the value of the products and the measure in which they contribute tot the realisation of goals.
From product quality to product function	Risk of describing qualities that are not related to a function and therefore difficult to measure.

Maintenance

Requirements are developed within a project to reach certain goals. After the project stops requirements are archived with all other project documents. When the project delivers software systems, these software systems are subsequently maintained by hierarchical departments. Maintaining mostly means to assure the functionality and qualities of the software. This sometimes involves expansions or changes to the software. In case of changes new projects are started. Should these projects develop the requirements all over again? If requirements are managed sufficiently after the delivery of projects, change projects and other projects can make good use of the already developed requirements. Goals are still known and rationales for certain solutions may still be valid and should provide valuable information for subsequent change projects.

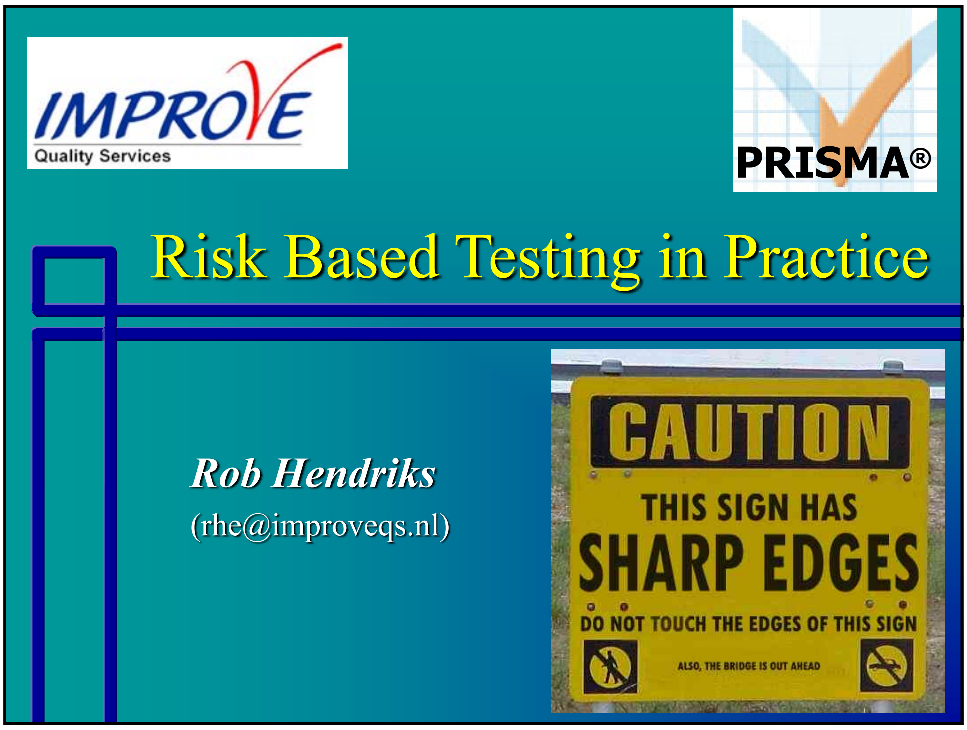
All this is even more complicated with todays growing complexity of softwareprograms.

Why do organisations not invest in requirements in general and qualities in particular?

With only the development of stakeholder and product functions, half of what may be the intention, is achievable. When the required attention is given to qualities, much more value is added tot development projects. Even more when this information is managed after the projects have finished and the results have been made available for maintance indicated projects. The main question to be asked is; “why do organisations not invest in these aspects of requirements management?”. With little cost up front it is likely to prevent having to spend additional resources in maintenance and rework at later dates.
I would like to explore the possible answers tot this question.

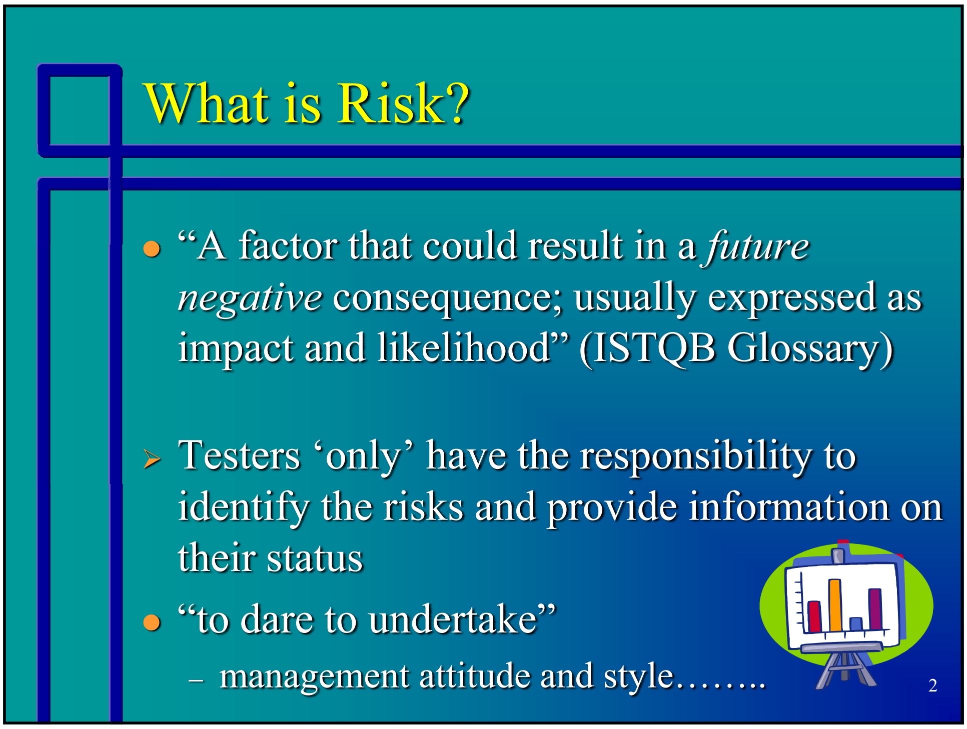
In the presentation examples will be presented to illustrate the above

Renze Zijlstra
Principle Consultant
KZA
www.kza.nl
06 2952 7225


IMPROVE
Quality Services
PRISMA®
CAUTION
THIS SIGN HAS
SHARP EDGES
DO NOT TOUCH THE EDGES OF THIS SIGN
ALSO, THE BRIDGE IS OUT AHEAD

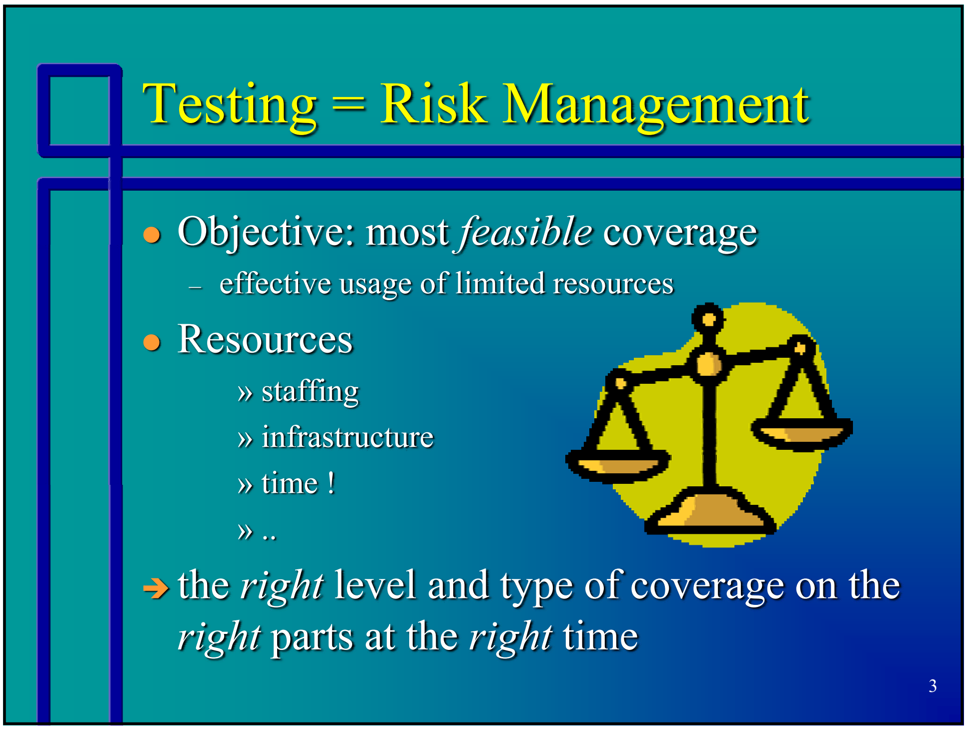
Risk Based Testing in Practice

Rob Hendriks
(rhe@improveqs.nl)



What is Risk?

- "A factor that could result in a *future negative* consequence; usually expressed as impact and likelihood" (ISTQB Glossary)
- Testers 'only' have the responsibility to identify the risks and provide information on their status
- "to dare to undertake"
  - management attitude and style……..

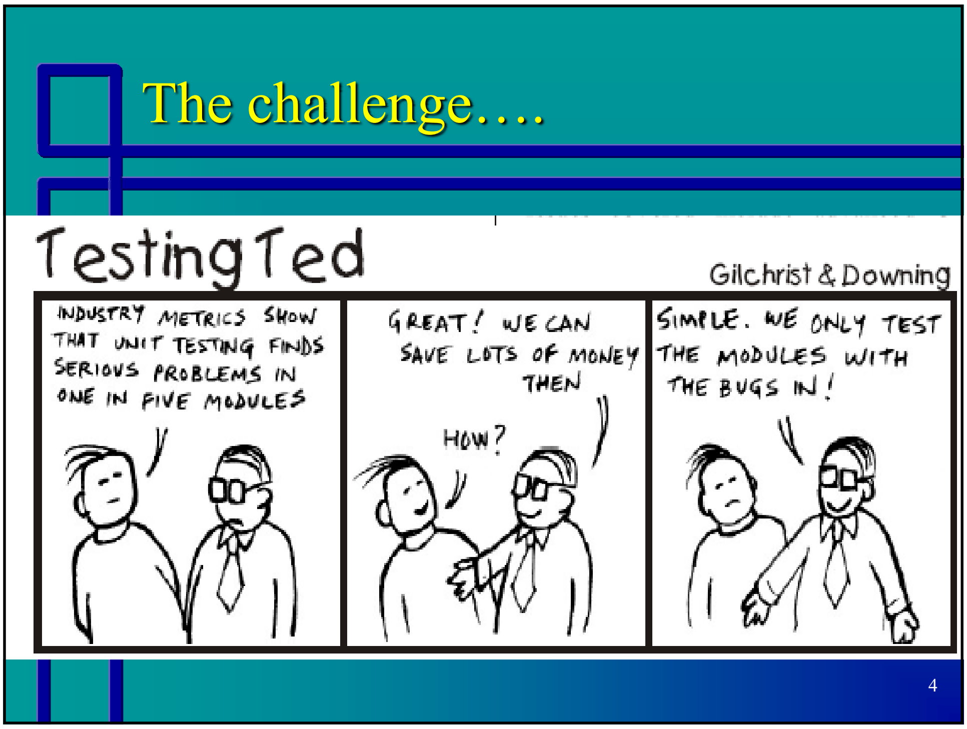


Testing = Risk Management

- Objective: most *feasible* coverage
  - effective usage of limited resources
- Resources
  - » staffing
  - » infrastructure
  - » time !
  - » ..

➔ the *right* level and type of coverage on the *right* parts at the *right* time

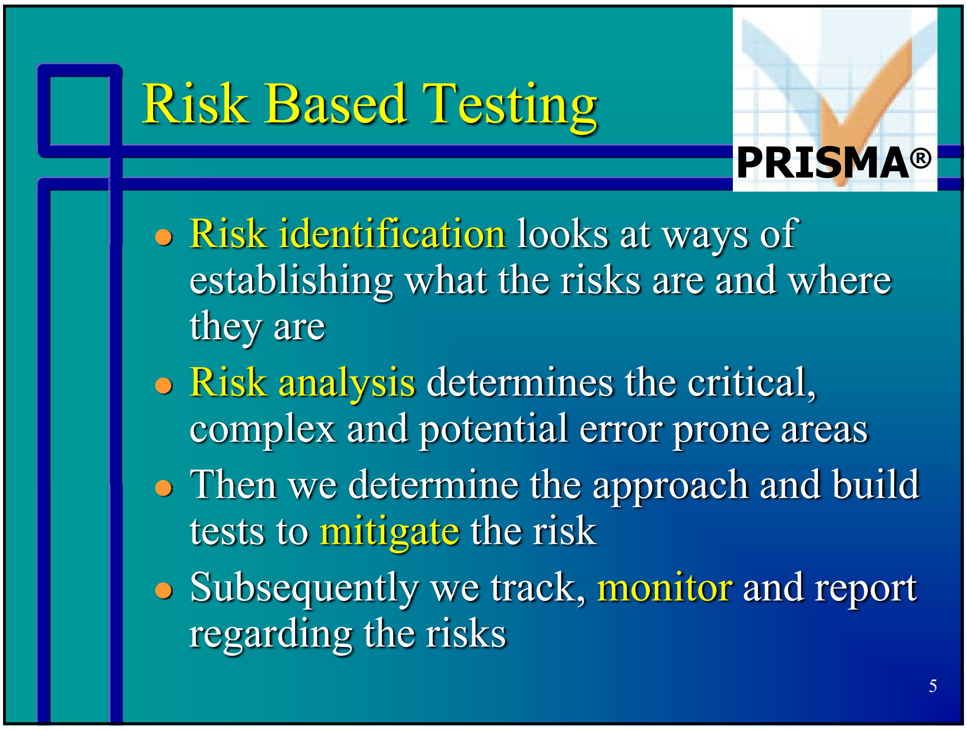
3



The challenge….

Testing Ted
Gilchrist & Downing
INDUSTRY METRICS SHOW THAT UNIT TESTING FINDS SERIOUS PROBLEMS IN ONE IN FIVE MODULES
GREAT! WE CAN SAVE LOTS OF MONEY THEN
HOW?
SIMPLE. WE ONLY TEST THE MODULES WITH THE BUGS IN!

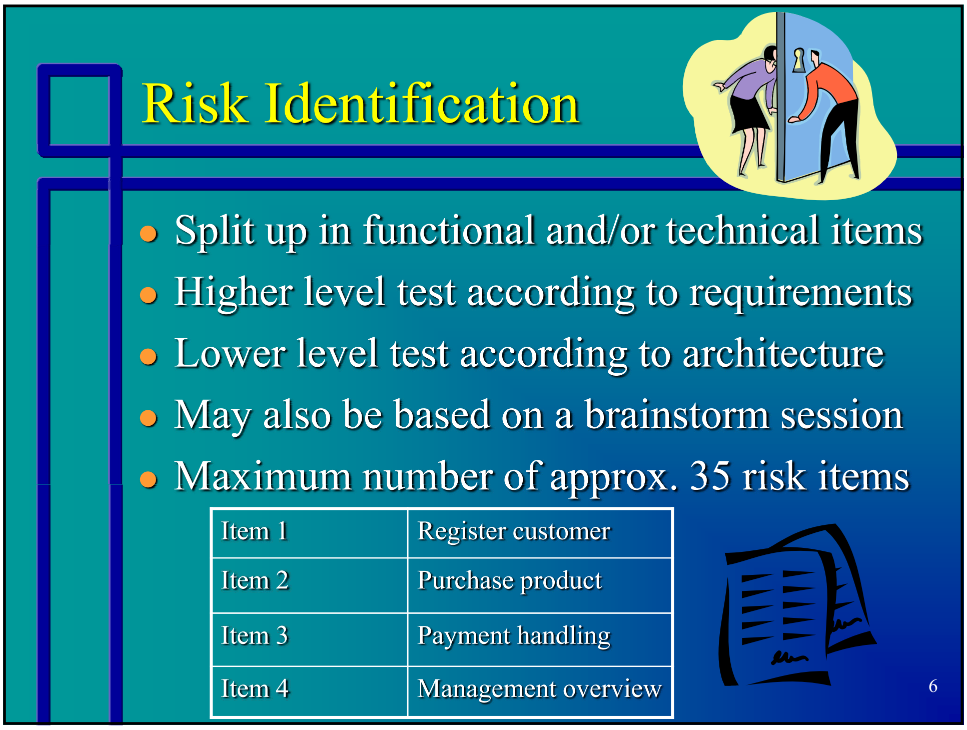
4



Risk Based Testing

PRISMA®

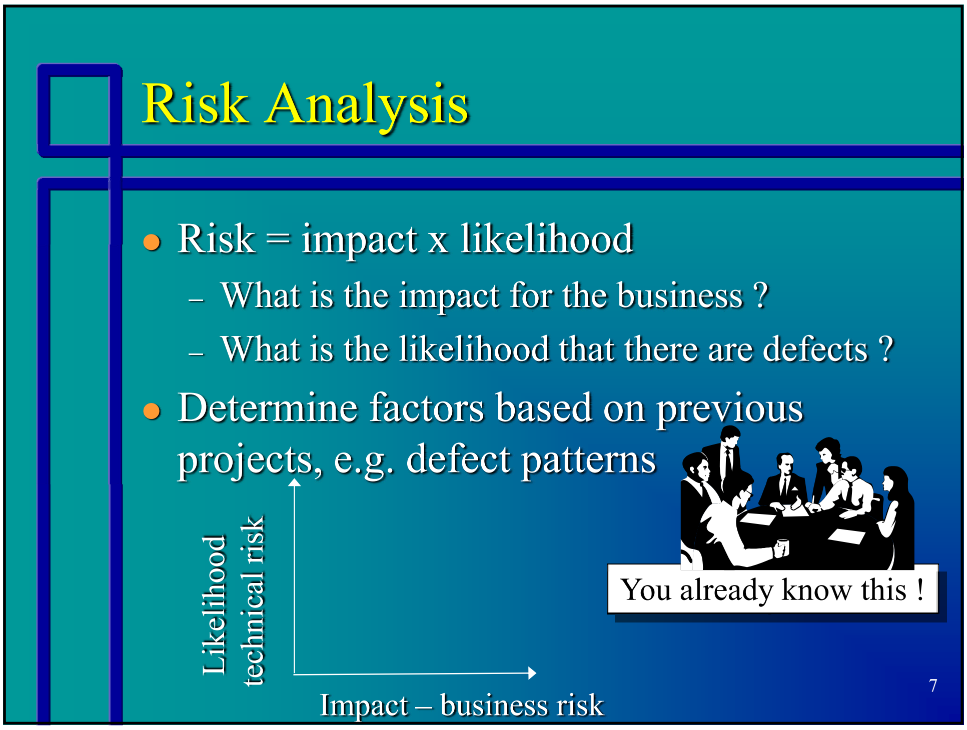
- Risk identification looks at ways of establishing what the risks are and where they are
- Risk analysis determines the critical, complex and potential error prone areas
- Then we determine the approach and build tests to mitigate the risk
- Subsequently we track, monitor and report regarding the risks



Risk Identification

- Split up in functional and/or technical items
- Higher level test according to requirements
- Lower level test according to architecture
- May also be based on a brainstorm session
- Maximum number of approx. 35 risk items

Item 1	Register customer
Item 2	Purchase product
Item 3	Payment handling
Item 4	Management overview



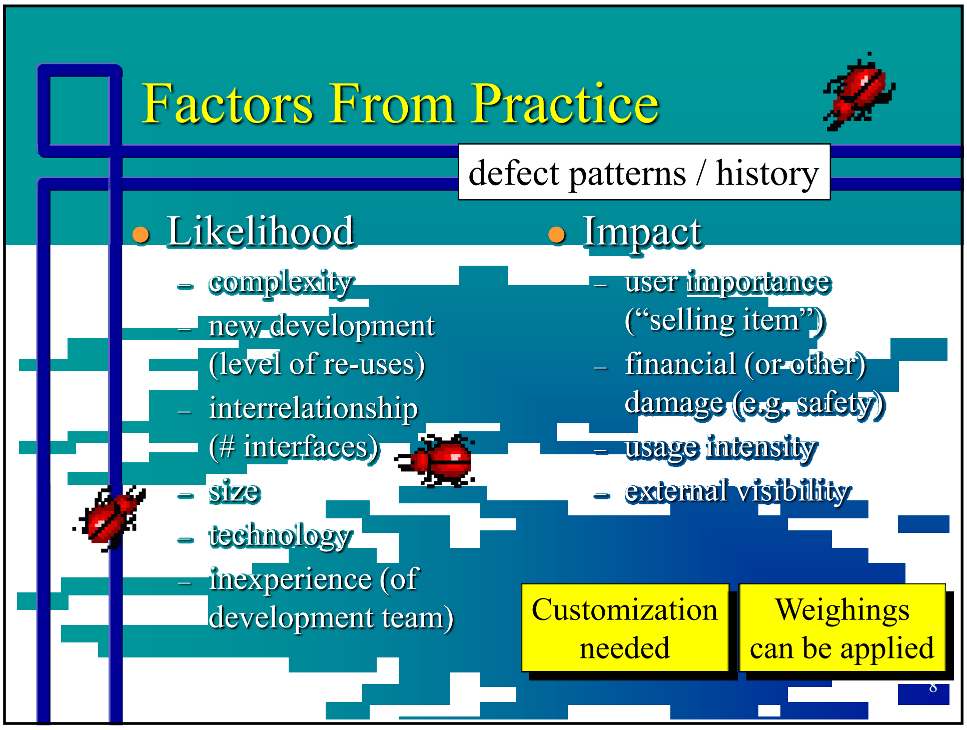
Risk Analysis

- Risk = impact x likelihood
  - What is the impact for the business ?
  - What is the likelihood that there are defects ?
- Determine factors based on previous projects, e.g. defect patterns

Likelihood
technical risk
Impact – business risk

You already know this !

7



Factors From Practice

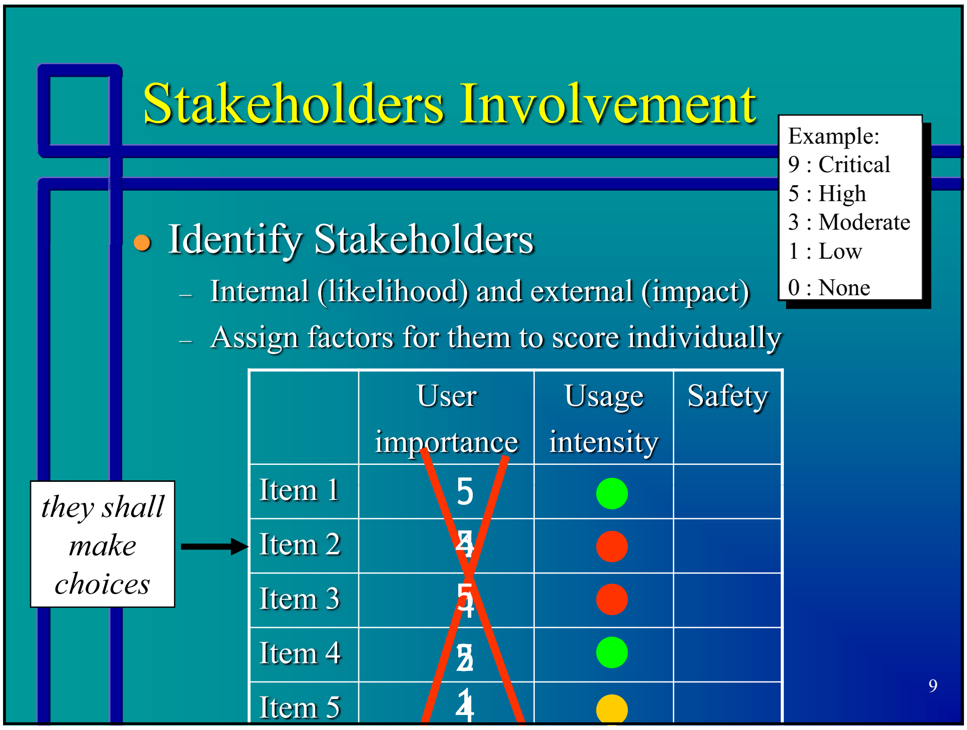
defect patterns / history

- Likelihood
  - complexity
  - new development (level of re-uses)
  - interrelationship (# interfaces)
  - size
  - technology
  - inexperience (of development team)
- Impact
  - user importance (“selling item”)
  - financial (or other) damage (e.g. safety)
  - usage intensity
  - external visibility

Customization needed

Weighings can be applied

8



Stakeholders Involvement

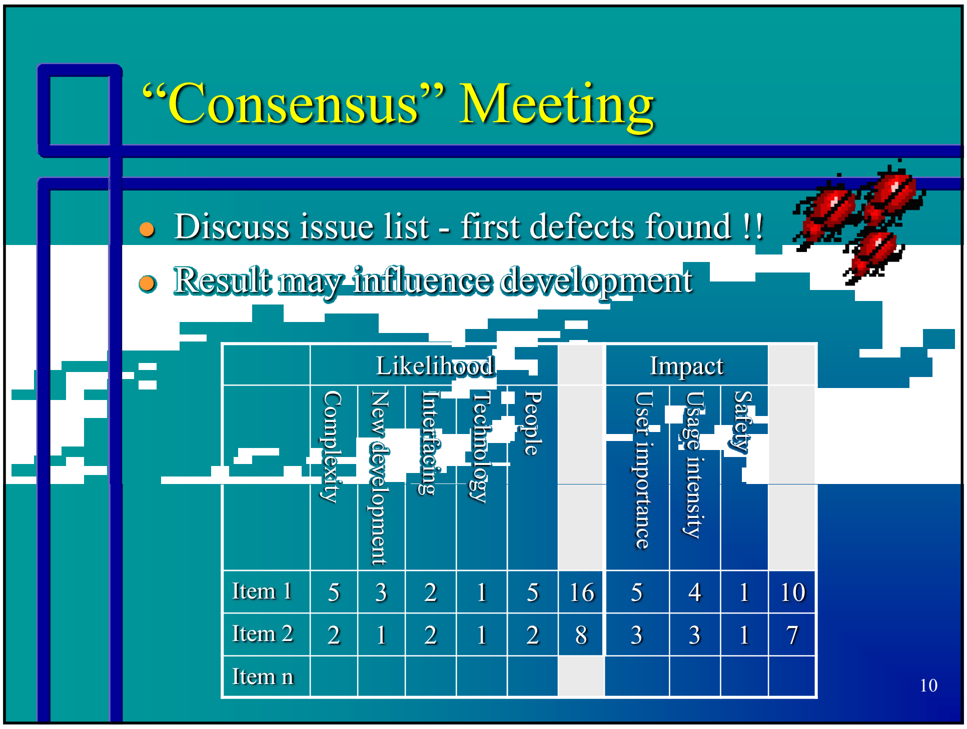
Example:
9 : Critical
5 : High
3 : Moderate
1 : Low
0 : None

- Identify Stakeholders
  - Internal (likelihood) and external (impact)
  - Assign factors for them to score individually

they shall make choices

	User importance	Usage intensity	Safety
Item 1	5	🟢	
Item 2	1 4	🔴	
Item 3	5 4	🔴	
Item 4	5 2	🟢	
Item 5	1 4	🟡	

9

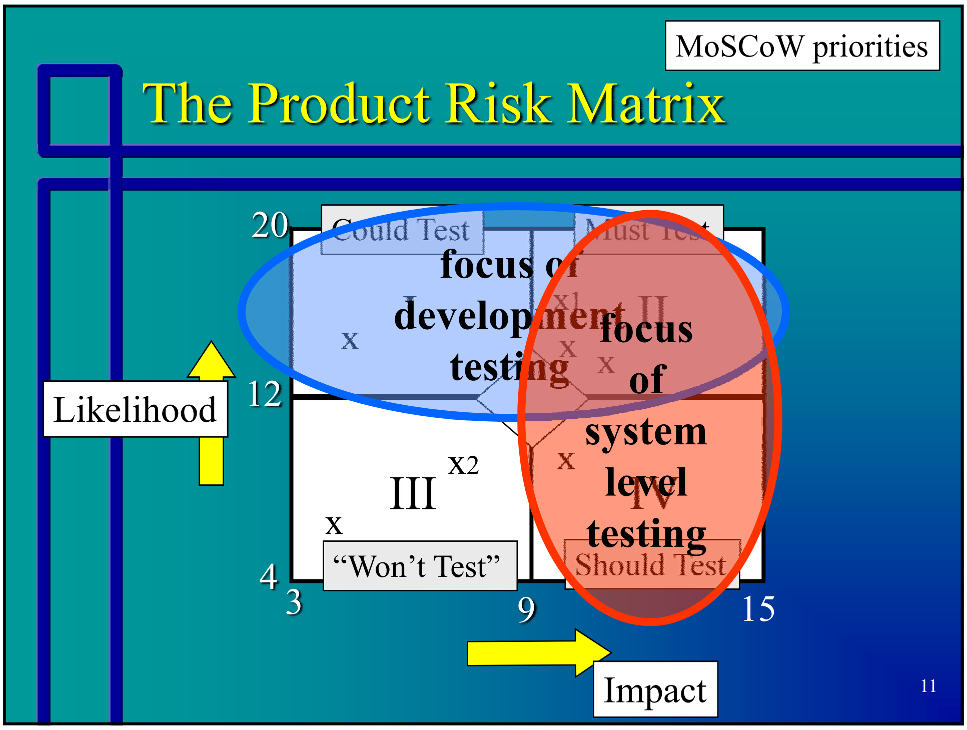


"Consensus" Meeting

- Discuss issue list - first defects found !!
- Result may influence development

	Likelihood						Impact			
	Complexity	New development	Interfacing	Technology	People		User importance	Usage intensity	Safety	
Item 1	5	3	2	1	5	16	5	4	1	10
Item 2	2	1	2	1	2	8	3	3	1	7
Item n										

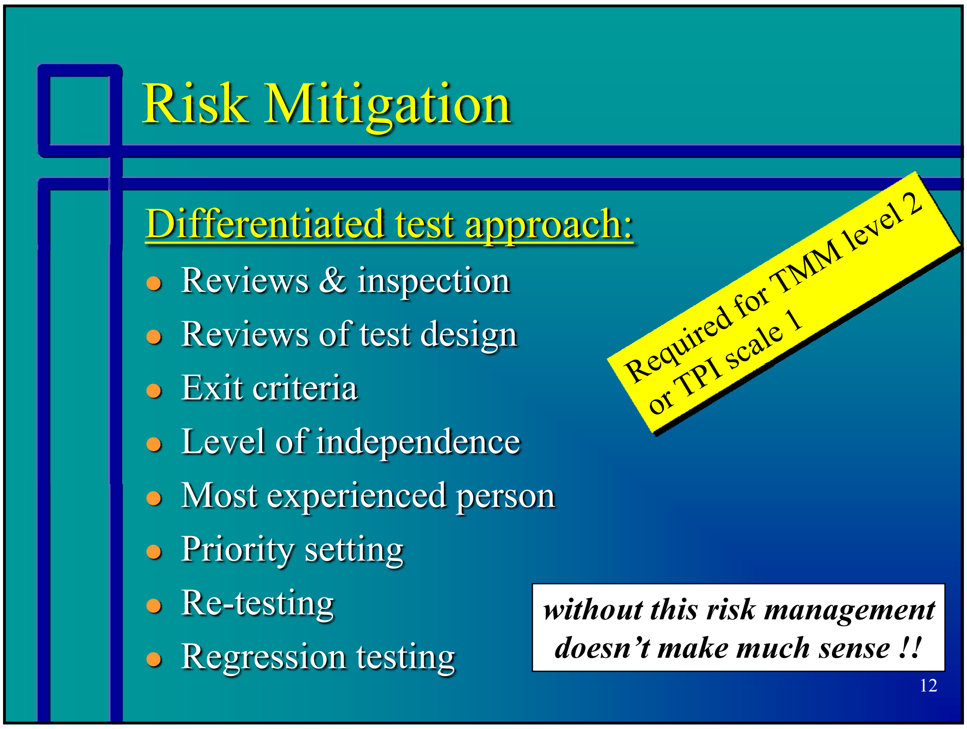
10



MoSCoW priorities

The Product Risk Matrix

11



Risk Mitigation

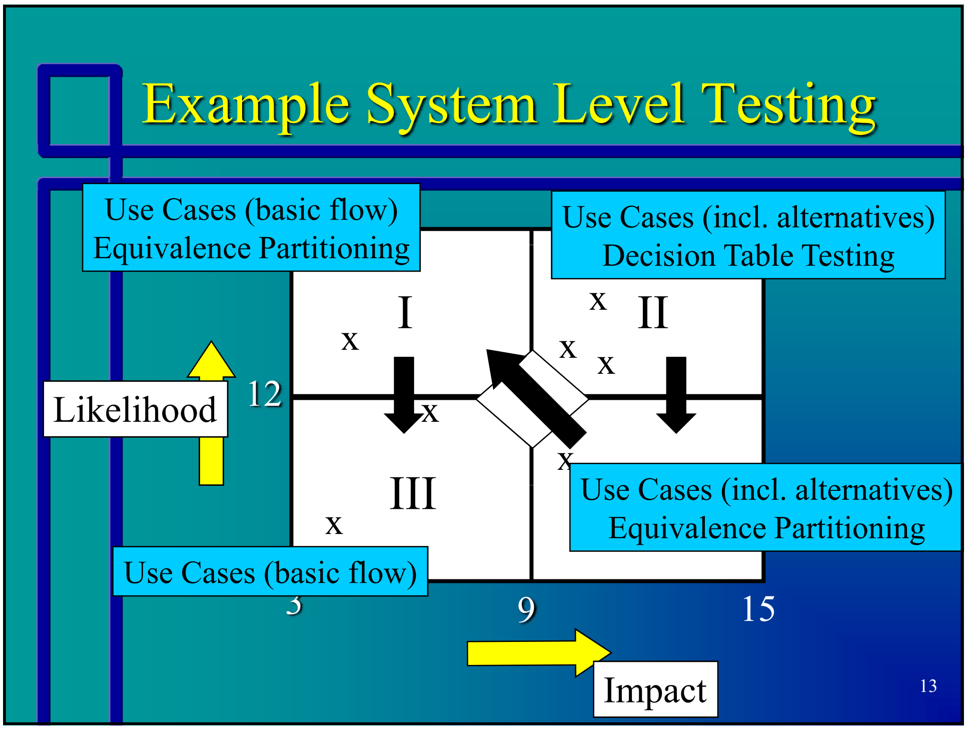
Differentiated test approach:

- Reviews & inspection
- Reviews of test design
- Exit criteria
- Level of independence
- Most experienced person
- Priority setting
- Re-testing
- Regression testing

Required for TMM level 2 or TPI scale 1

without this risk management doesn't make much sense !!

12



Example System Level Testing

Use Cases (basic flow)
Equivalence Partitioning

Use Cases (incl. alternatives)
Decision Table Testing

I

II

III

Likelihood

12

Use Cases (incl. alternatives)
Equivalence Partitioning

Use Cases (basic flow)

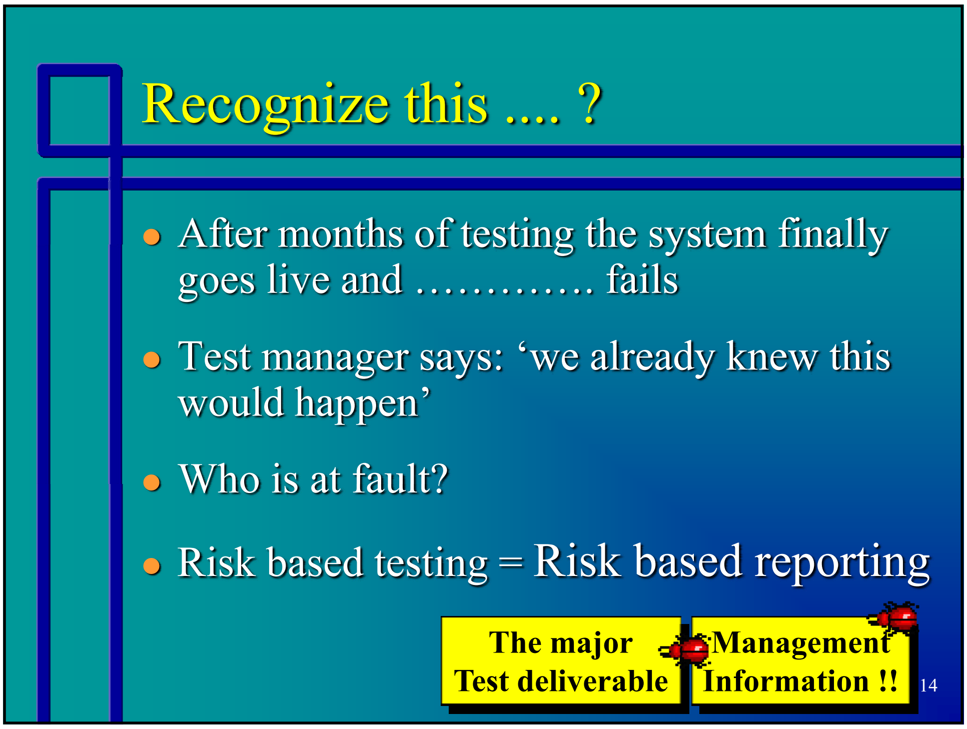
3

9

15

Impact

13



Recognize this .... ?

- After months of testing the system finally goes live and …………. fails
- Test manager says: ‘we already knew this would happen’
- Who is at fault?
- Risk based testing = Risk based reporting

The major
Test deliverable

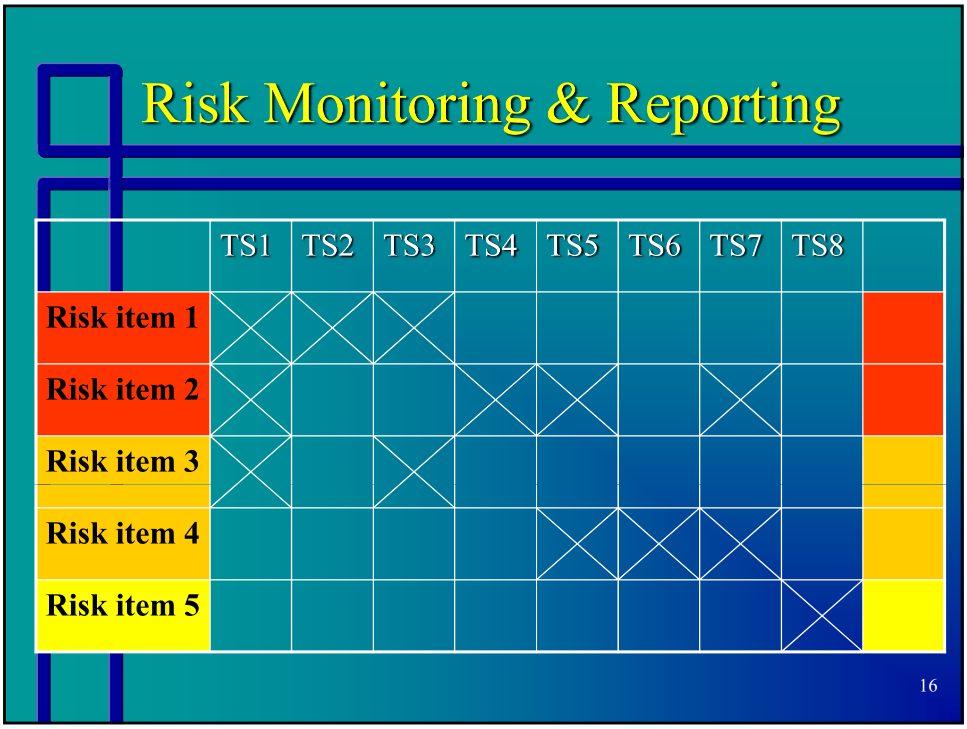
Management
Information !!

14

Communication Levels …

Testing Ted
Gilchrist & Downing
I WANT YOU TO GET ME SOME METRICS ABOUT THIS CODE, PLEASE TED
WHAT WOULD YOU LIKE? HALSTEAD_PURITY RATIO? MOOSE_COUPLING_BETWEEN OBJECTS?
NO, JUST DOES_IT_WORK ?!

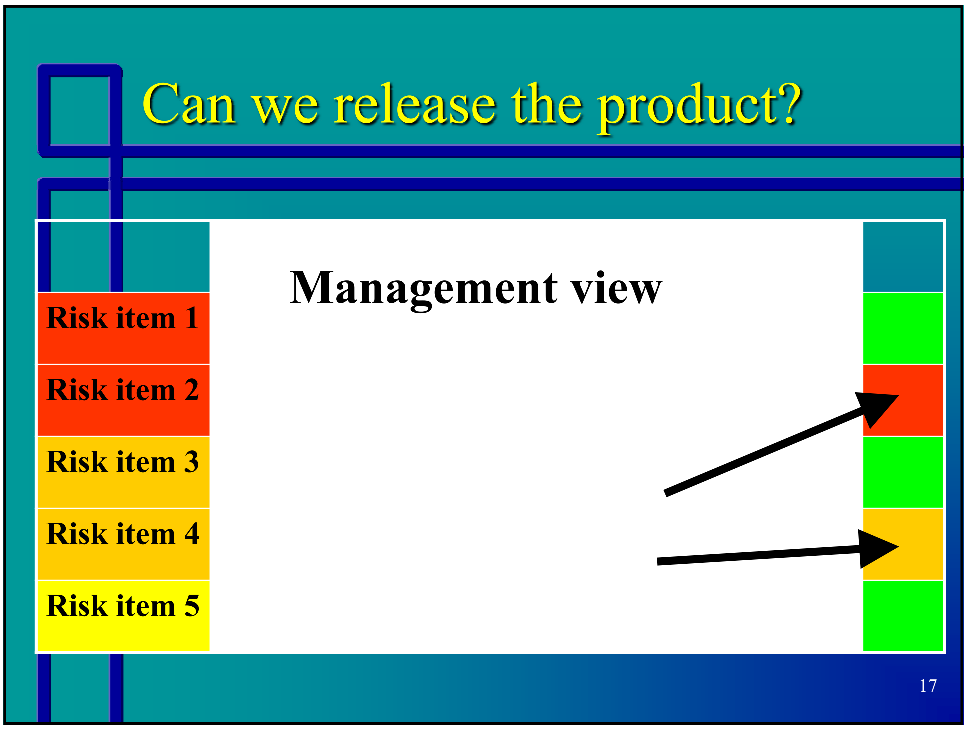
15



Risk Monitoring & Reporting

	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	
Risk item 1	☒	☒	☒						🟥
Risk item 2	☒			☒	☒		☒		🟥
Risk item 3	☒		☒						🟧
Risk item 4					☒	☒	☒		🟧
Risk item 5								☒	🟨

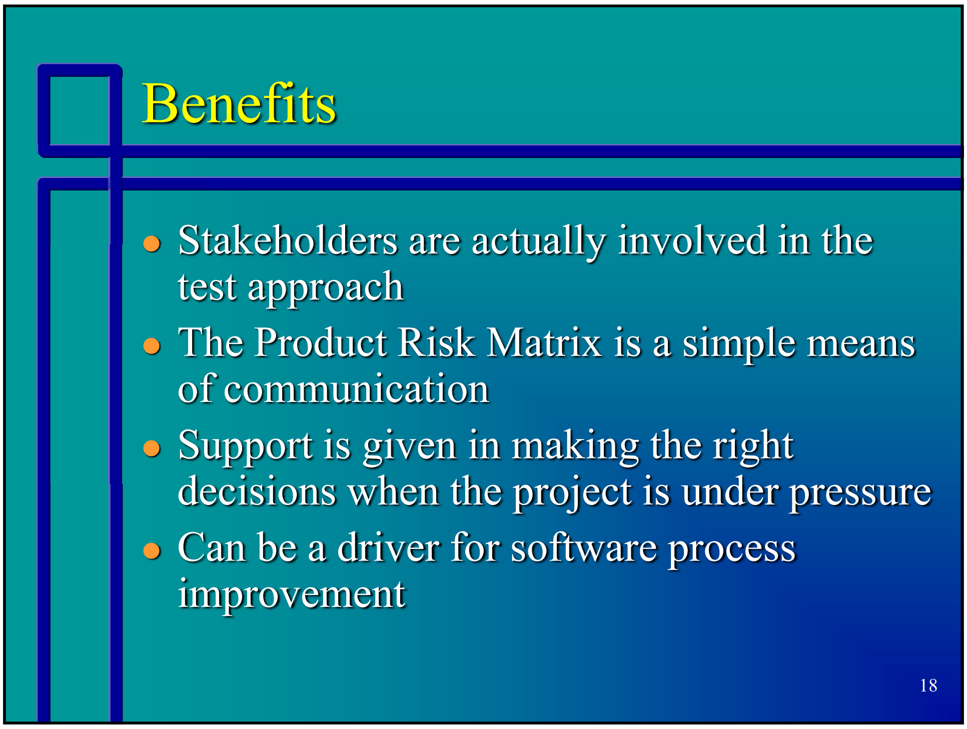
16



Can we release the product?

Management view

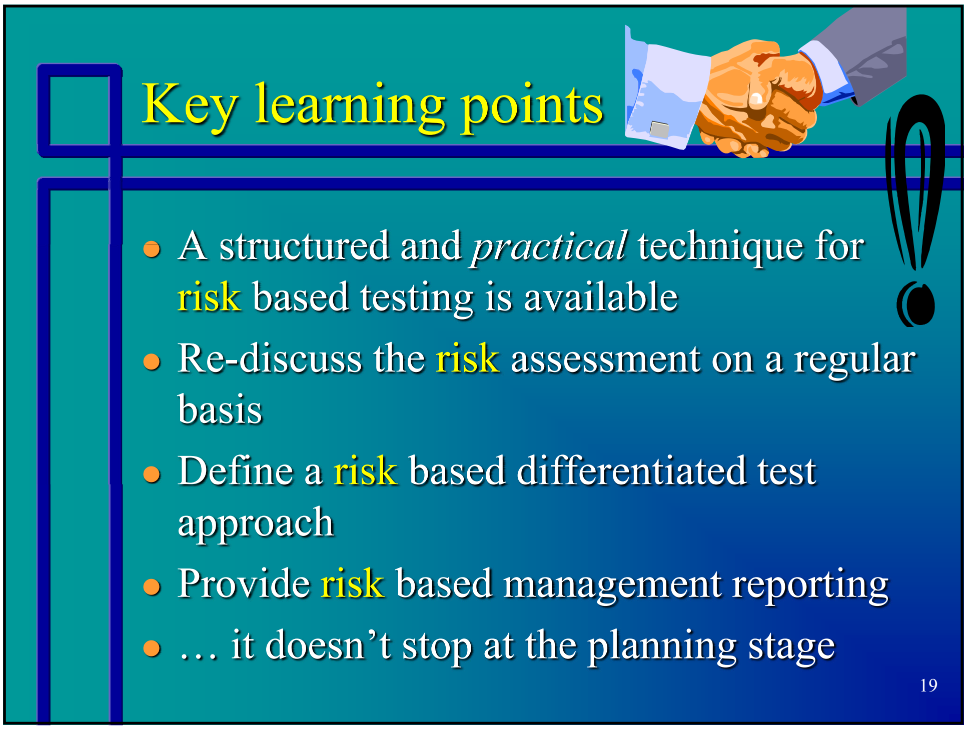
17



Benefits

- Stakeholders are actually involved in the test approach
- The Product Risk Matrix is a simple means of communication
- Support is given in making the right decisions when the project is under pressure
- Can be a driver for software process improvement

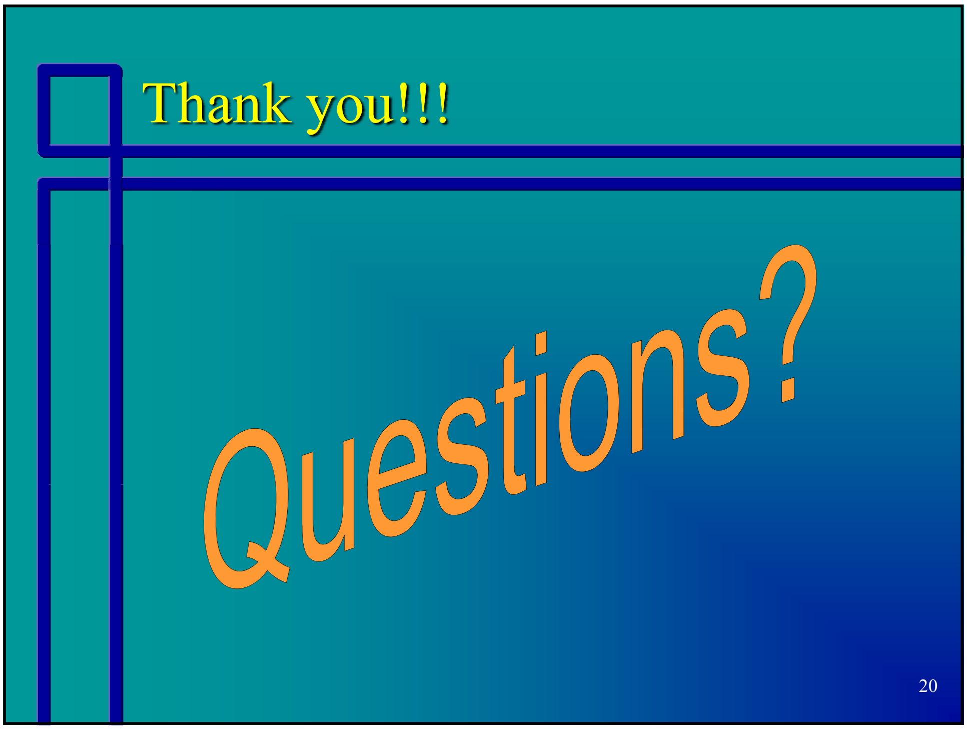
18



Key learning points

- A structured and *practical* technique for risk based testing is available
- Re-discuss the risk assessment on a regular basis
- Define a risk based differentiated test approach
- Provide risk based management reporting
- … it doesn't stop at the planning stage

19



Thank you!!!

Questions?

20

A New Statistical Software Reliability Tool

M.A.A. Boon[1], E. Brandt[2], I. Corro Ramos[1], A. Di Bucchianico[1] and R. Henzen[2]

[1] *Department of Mathematics, Eindhoven University of Technology, Eindhoven, The Netherlands*

[2] *Refis System Reliability Engineering, Bilthoven, The Netherlands*

Abstract

We describe a new statistical tool for software reliability analyses that we are developing. Existing packages for statistical analysis of software reliability data do not make full use of state-of-the-art statistical methodology or do not conform to best practices in statistics. Our tool has a Java based interface and uses the statistical programming language R (see www.r-project.org) for the statistical computations. R is open-source free software maintained by a group of top-level statisticians and is rapidly becoming the standard programming language within the statistical community. The tool has a user-friendly interface which includes features like auto detection of data type and a model selection wizard.

Keywords: software reliability, software testing, statistical models, R.

1 Introduction

Successful testing processes require excellence in both software testing and management. In order to support well-founded decisions on issues like resource allocation and software release moments, quantitative procedures are indispensable. Since few testing processes have a deterministic course, statistics is very often an appropriate part of such quantitative procedures. Existing tools for software reliability analysis like Casre and Smerfs[3] do not make full use of state-of-the-art statistical methodology or do not conform to best practices in statistics. Thus, these tools cannot fully support sound software reliability analyses. We decided to build a new tool that

- uses well-documented state-of-the-art algorithms
- is platform independent
- encourages to apply best practices from statistics
- can easily be extended to incorporate new models.

In order to meet these requirements we decided to use Java for the interface and the statistical programming language R (see www.r-project.org) for the statistical computations. R is open-source free software maintained by a group of top-level statisticians and is rapidly becoming the standard programming language within the statistical community. In this paper we report on the status of our tool. Our tool is a joint project of the Laboratory for Quality Software (LaQuSo) of the Eindhoven University of Technology (www.laquso.com) and Refis (www.refis.nl). The tool development is financially supported by a grant of the Dutch Innovation Platform.

The rest of the paper is organized as follows. In Section 2 general guidelines of software reliability analysis are given. In Section 3 we present general and statistical features of the tool. We focus on describing the tool's GUI thoroughly. A study to show how the tool works in practice is presented in Section 4. Finally, in Section 5 we summarize the work carried out and we point out to several important short-term objectives.

2 A note on reliability analysis

For general information on software reliability analysis we refer to Lyu (1996), Musa (2006) and Pham (2006). Like there exist coding standards for writing software, there also exist standards for performing statistical analyses. Basic steps in a statistical analysis of software reliability data should include (cf. Goel (1985))

1. data collection (which data is relevant for the analysis)
2. trend tests (does the data indicate growth, otherwise analysis is useless)
3. model selection (pre-selection of models)
4. model estimation (calculate optimal parameters from data)
5. model validation (do models fit to data)
6. model interpretation (calculate quantities of interest from model parameters).

We talk of ungrouped or exact data when the failures are reported individually and the data represents time between failures. However, it is possible to report faults in periods of time, in which case the data consists of the time intervals where the failures are reported and the number or faults found in each interval. In this case we talk of grouped or interval data. Unfortunately, ungrouped data has received much more attention in the literature.

Before trying to fit any reliability growth model we should verify whether the data indicates reliability growth. This can be done using adequate plots or more formally with trend tests. Figure 1 clearly depicts the idea that software becomes more reliable as long as it is tested and errors have been repaired, so that more effort is required to find future errors.

There are over 200 software reliability models based on different assumptions, assumptions which are often unclear or too unrealistic. Systematic approaches to use model assumptions and data requirements for initial model selection have not received much attention in the literature, Kharchenko et al. (2002) being an exception. Therefore, we have been developing a matrix-based procedure to support the choice of the models to work with. A simple version of this matrix can be found in Figure 2. Since we wish to select rather than rule out applicable models, we state all assumptions as negations of restrictions. To select models, one first has to select relevant assumptions and weights to incorporate the available information on the testing project at hand. If all requirements and selected assumptions of a model are satisfied, then the score for this model is 100%. In all other cases the score of the model is defined using the relative importance (weight) of the applicable characteristics of the model.

Estimation of model parameters requires optimization. Since the parameters typically are of different order of magnitude, numerical problems like non-convergence or large flat areas

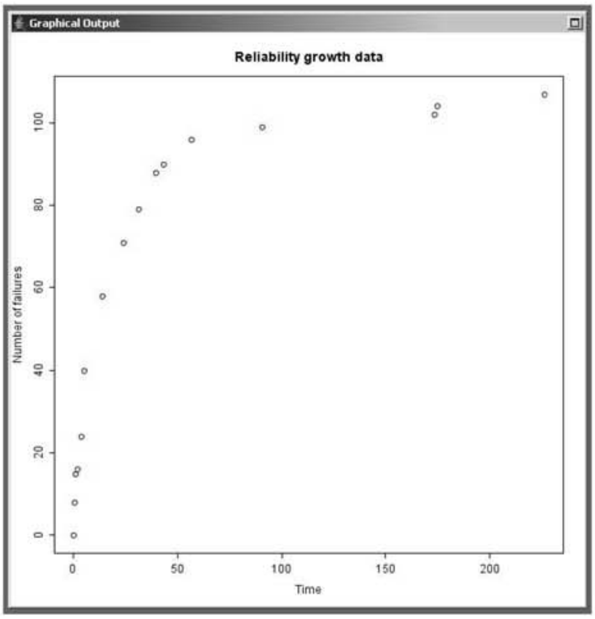


Figure 1: Reliability growth data.

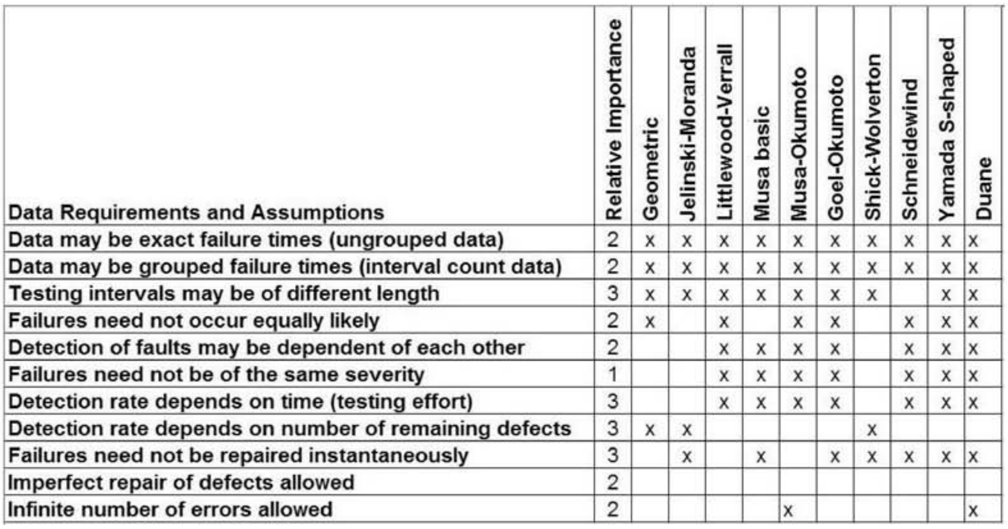
around the maximum cause practical problems (see Yin and Trivedi (1999)). In our tool we pay attention to convergence issues and apply algorithms that avoid the standard numerical problems.

After parameter estimation has been completed, model verification must be determined. Graphical methods like the u-plot or TTT (Total Time on Test) plots (see Rigdon and Basu (2000) for details) or goodness-of-fit tests can be applied. Unluckily we find also problems here (derived to the fact that the assumptions of independent and identically distributed observations are normally broken by software reliability models). Therefore, standard goodness-of-fit tests, like the Kolmogorov test, cannot be used although this is often done. For diverse subclasses of models new goodness-of-fit tests are becoming available (see Bhattacharjee et al. (2004), Zhao and Wang (2005)).

Finally, model interpretation makes possible to determine quantities like number of remaining errors and reliability of the system.

3 Software reliability tool

In this section we give a general overview of the GUI of our tool. Our tool is written in Java and we have made use of readily available components from Java Resource Bundles. The statistical computations are performed by calling R (high-quality free open-source statistical software). The communication of Java with R uses JRI and JavaGD libraries developed by RoSuDa, the Computational Statistics group of the University of Augsburg. Initially, the GUI of our tool consists of four menu items as we can see in Figure 3. The multiple options



Data Requirements and Assumptions	Relative Importance	Geometric	Jelinski-Moranda	Littlewood-Verrall	Musa basic	Musa-Okumoto	Goel-Okumoto	Shick-Wolverton	Schneidewind	Yamada S-shaped	Duane
Data may be exact failure times (ungrouped data)	2	x	x	x	x	x	x	x	x	x	x
Data may be grouped failure times (interval count data)	2	x	x	x	x	x	x	x	x	x	x
Testing intervals may be of different length	3	x	x	x	x	x	x	x		x	x
Failures need not occur equally likely	2	x		x		x	x		x	x	x
Detection of faults may be dependent of each other	2			x	x	x	x		x	x	x
Failures need not be of the same severity	1			x	x	x	x		x	x	x
Detection rate depends on time (testing effort)	3			x	x	x	x		x	x	x
Detection rate depends on number of remaining defects	3	x	x					x			
Failures need not be repaired instantaneously	3		x		x		x	x	x	x	x
Imperfect repair of defects allowed	2										
Infinite number of errors allowed	2					x					x

Figure 2: Assumption matrix.

of all the menus are explained during this section. In the beginning only the *Data* and *Help* menus are enabled. To have access to the working environment we have to select the option *Import* from the *Data* menu. We can distinguish two new windows that will remain visible all the time, one devoted to data and the other one to graphics. Figure 4 shows the data window. Three different tabs can be identified. The first one contains the imported data, the second one is allocated to the filtered data (in case we use this option) and the third one will show the data from the model analysis. Figure 1 shows the *Graphical Output* window where all the plots produced will be displayed. In the rest of this section we explain the tool menu options in detail, paying special attention to the *Models* and *Analysis* menus.

3.1 Data menu

1. **Import**

   Imports data files in .txt and .xls formats. After loading the data set the working environment is displayed.

2. **Export**

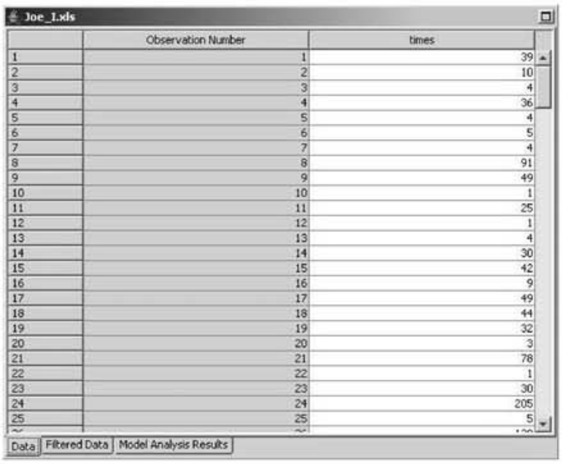
   Exports data files in .txt and .xls formats.

3. **Filter Data**

   This dialog (see Figure 5) allows to refine the data set and keep a subset of observations. The filtered data set can also be exported and imported.

Refis and LaQuSo Tool
Data
Graphics
Analysis
Help
Refis
system reliability engineering
TU/e
technische universiteit eindhoven
LaQuSo
SRE Tool
© 2006 Technische Universiteit Eindhoven, LaQuSo and Refis

Figure 3: Software reliability tool GUI.



Joe_1.xls

	Observation Number	times
1	1	39
2	2	10
3	3	4
4	4	36
5	5	4
6	6	5
7	7	4
8	8	91
9	9	49
10	10	1
11	11	25
12	12	1
13	13	4
14	14	30
15	15	42
16	16	9
17	17	49
18	18	44
19	19	32
20	20	3
21	21	78
22	22	1
23	23	30
24	24	205
25	25	5

Figure 4: Data window.

4. **Transform Variable**

   With this option we can create new variables applying basic operations to the variables of the data set. The available operations can be observed in Figure 6.

5. **Preferences**

   User preferences can be set using this option. Font type and size can be chosen for both program GUI and output (see Figure 7). The number of significant digits for the calculations can be selected from one up to eight (by default set to three). We can also

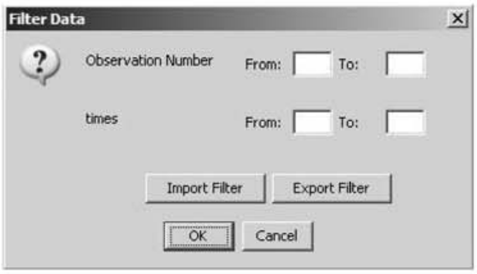


Figure 5: Filter Data dialog.

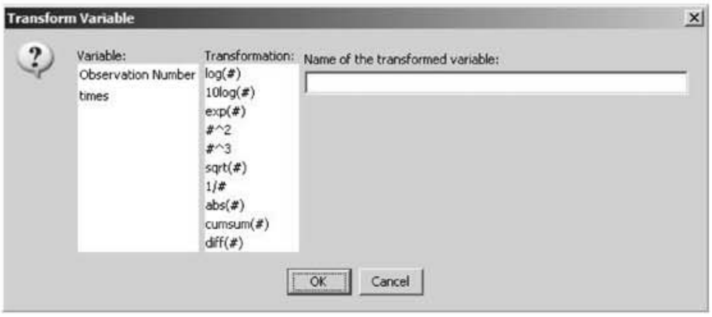


Figure 6: Transform Variable dialog.

decide the colours to be used in the plots. These options can be saved (or load) to (from) a .txt file.

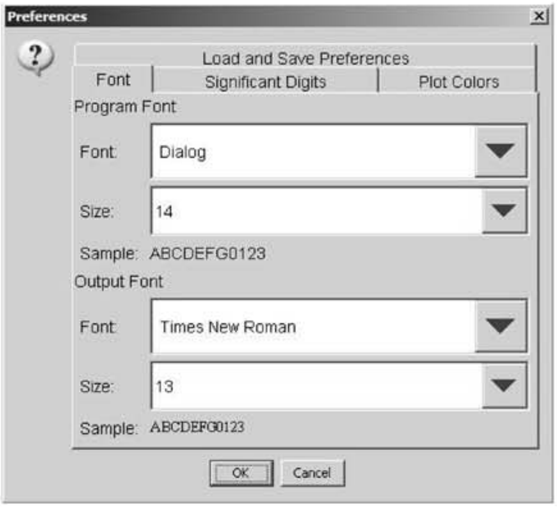
6. **Exit**

   Closes the GUI and terminates the program execution.

3.2 Graphics menu

1. **Create Scatter Plot**

   With this option we can generate plots that will be loaded in the *Graphical Output* window. After clicking on this item, a window named *Select Plot Variables* pops-up (see Figure 8). We can select the variables we want to plot and assign them to the X or Y axes. Moreover, we can choose the type of plot we wish to generate. The existing options are scatter plot, normal probability plot and TTT plot. Finally, the *Draw Graph* button will create the graph. On the lower part of the window we can set



Program Font

Font: Dialog

Size: 14

Sample: ABCDEFG0123

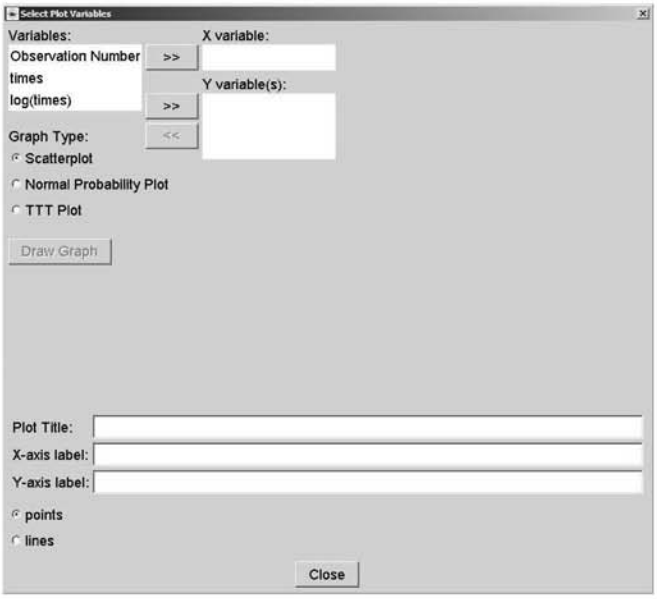
Output Font

Font: Times New Roman

Size: 13

Sample: ABCDEFG0123

Figure 7: Preferences window.



Variables:

Observation Number
times
log(times)

X variable:

Y variable(s):

Graph Type:

● Scatterplot

○ Normal Probability Plot

○ TTT Plot

Plot Title:

X-axis label:

Y-axis label:

● points

○ lines

Figure 8: Select Plot Variables window.

display options of the plot like the main title, the labels of the axes or the format of the curves.

2. **Copy Current Plot to Clipboard**

   Copy the plot that is displayed on the *Graphical Output* window to the clipboard. Thus, the plot is cached and can be transferred between documents or applications, via copy and paste operations.

3. **Export Current Plot**

   Selecting this option a save dialog appears to export the current plot to a graphic file. The formats supported are .eps, .jpg and .png.

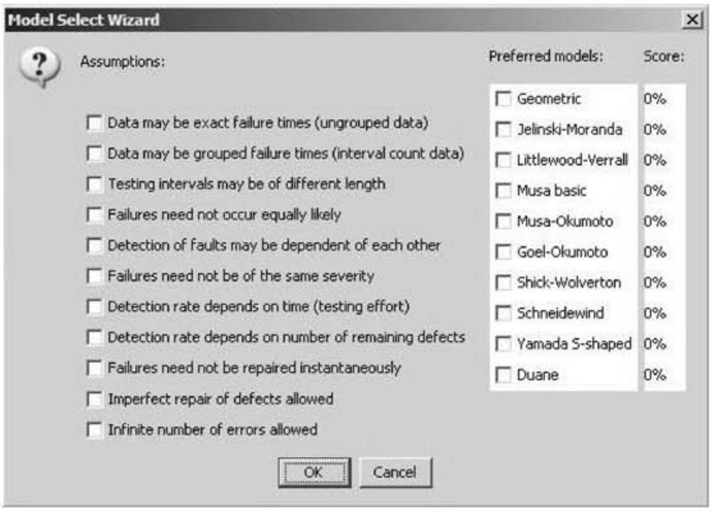
4. **Print Current Plot**

   With this option a print dialog is displayed to print the current plot.

3.3 Analysis menu

1. **Model Select Wizard**

   Figure 9 shows the *Model Select Wizard*. In this dialog a list of common software reliability assumptions are enumerated. These assumptions are known from the literature and they are used in different software reliability models. The most relevant models are listed on the right-hand side of the dialog follow by a column called *Score*. Every assumption has a score associated to every model. After selecting the assumptions, the models receive points and they are sorted by relevance. The higher score a model has, the better the model will fit the assumptions.



Assumptions:

☐ Data may be exact failure times (ungrouped data)
☐ Data may be grouped failure times (interval count data)
☐ Testing intervals may be of different length
☐ Failures need not occur equally likely
☐ Detection of faults may be dependent of each other
☐ Failures need not be of the same severity
☐ Detection rate depends on time (testing effort)
☐ Detection rate depends on number of remaining defects
☐ Failures need not be repaired instantaneously
☐ Imperfect repair of defects allowed
☐ Infinite number of errors allowed

Preferred models:	Score:
☐ Geometric	0%
☐ Jelinski-Moranda	0%
☐ Littlewood-Verrall	0%
☐ Musa basic	0%
☐ Musa-Okumoto	0%
☐ Goel-Okumoto	0%
☐ Shick-Wolverton	0%
☐ Schneidewind	0%
☐ Yamada S-shaped	0%
☐ Duane	0%

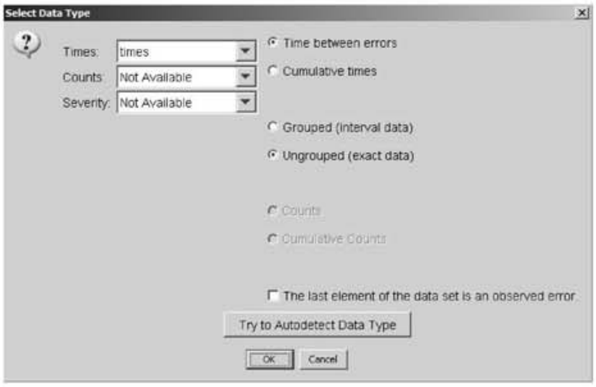
Figure 9: Model Select Wizard dialog.

2. **Data Type**

   With the *Select Data Type* window (see Figure 10) we have the option to set the type of data of our data set. Time between failures or cumulative times, exact or interval data are the available options. In case of interval data we have to distinguish between counts per interval or cumulative counts. Options like different kind of errors (severity) or whether the last element of the data set is an observer error are also offered. In case we do not know which kind of data we have, the option *Try to Autodetect Data Type* results of special interest. With this button the program estimates the type of data of our data file. Checking the number of columns (grouped or ungrouped) or the growth of the data (time between failures or cumulative times) we may find out what kind of data we are using.

3. **Trend Tests**

   The tool has the Laplace and the MIL-HDBK 189 trend tests already implemented and new tests will be added soon. The *Trend Tests* window is shown in Figure 11. Select



Select Data Type

Times: times

Counts: Not Available

Severity: Not Available

● Time between errors

○ Cumulative times

○ Grouped (interval data)

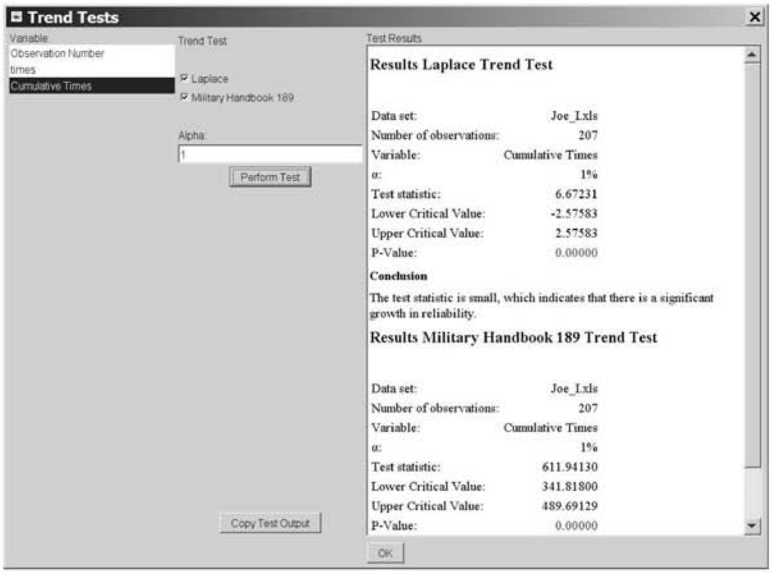
● Ungrouped (exact data)

○ Counts

○ Cumulative Counts

□ The last element of the data set is an observed error.

Figure 10: Select Data Type dialog.



Trend Tests

Variable: Observation Number / times / Cumulative Times (selected)

Trend Test: ☑ Laplace ☑ Military Handbook 189

Alpha: 1

Test Results

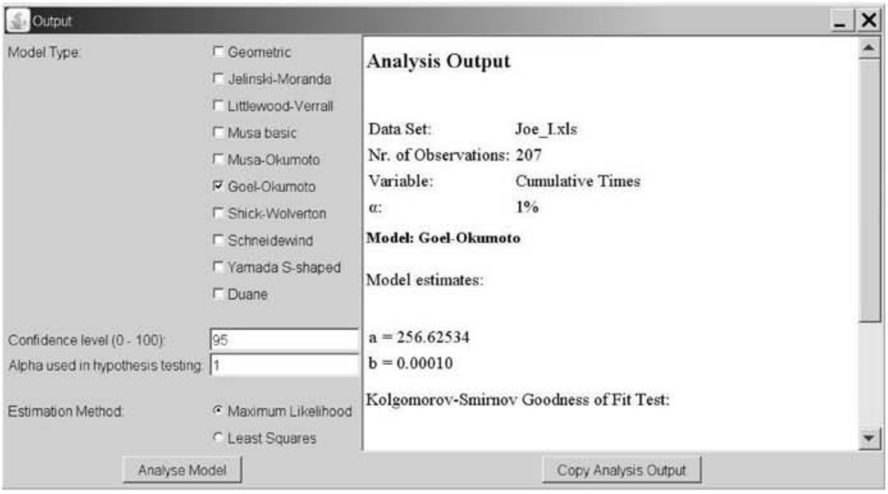
```
Results Laplace Trend Test

Data set:                      Joe_I.xls
Number of observations:              207
Variable:                 Cumulative Times
α:                                    1%
Test statistic:                  6.67231
Lower Critical Value:           -2.57583
Upper Critical Value:            2.57583
P-Value:                         0.00000

Conclusion
The test statistic is small, which indicates that there is a significant
growth in reliability.

Results Military Handbook 189 Trend Test

Data set:                      Joe_I.xls
Number of observations:              207
Variable:                 Cumulative Times
α:                                    1%
Test statistic:                611.94130
Lower Critical Value:          341.81800
Upper Critical Value:          489.69129
P-Value:                         0.00000
```

Figure 11: Trend Tests window.

the variable whose growth we want to investigate as well as the tests we would like to use. The significance at which the test will be performed can also be selected here. On the right-hand side of the window we can observe the result of the tests we decided to perform. Graphical interpretation of the test is also displayed on the *Graphical Output* window.

4. **Analyse Model**

   The analysis window is shown in Figure 12. Before performing the analysis four steps must be follow. First, select the model(s) to fit the data. Then set the confidence level used to calculate confidence intervals (by default set to 95%) and the significance level to perform hypothesis testing (by default set to 1%). Last, choose between the maximum-likelihood or least squares methods for parameter estimation. To perform



Model Type:
□ Geometric
□ Jelinski-Moranda
□ Littlewood-Verrall
□ Musa basic
□ Musa-Okumoto
☑ Goel-Okumoto
□ Shick-Wolverton
□ Schneidewind
□ Yamada S-shaped
□ Duane

Confidence level (0 - 100): 95
Alpha used in hypothesis testing: 1

Estimation Method:
● Maximum Likelihood
○ Least Squares

Analysis Output

Data Set: Joe_I.xls
Nr. of Observations: 207
Variable: Cumulative Times
α: 1%

Model: Goel-Okumoto

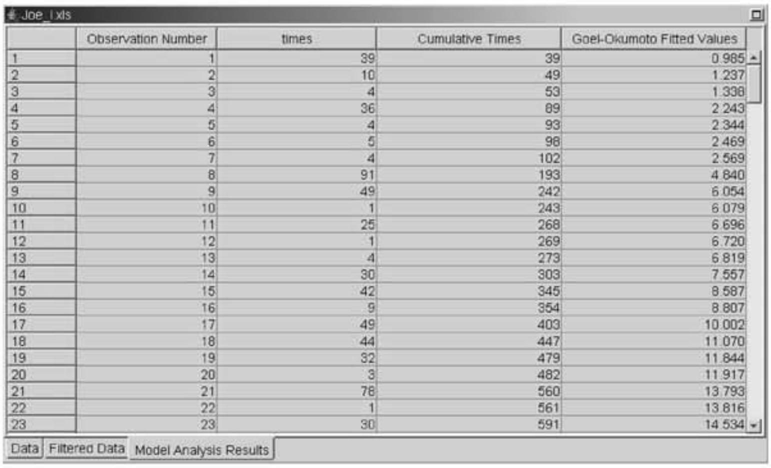
Model estimates:

a = 256.62534
b = 0.00010

Kolgomorov-Smirnov Goodness of Fit Test:

Figure 12: Analysis output.

the analysis click the *Analyse Model* button. Subsequently, the analysis output shows the type of data, the model choices, the parameter estimates and the result of the Kolmogorov goodness-of-fit test. The data window (see Figure 13) presents the fitted values for the selected models. Finally, the graphical output (see Figure 14) shows the



	Observation Number	times	Cumulative Times	Goel-Okumoto Fitted Values
1	1	39	39	0.985
2	2	10	49	1.237
3	3	4	53	1.338
4	4	36	89	2.243
5	5	4	93	2.344
6	6	5	98	2.469
7	7	4	102	2.569
8	8	91	193	4.840
9	9	49	242	6.054
10	10	1	243	6.079
11	11	25	268	6.696
12	12	1	269	6.720
13	13	4	273	6.819
14	14	30	303	7.557
15	15	42	345	8.587
16	16	9	354	8.807
17	17	49	403	10.002
18	18	44	447	11.070
19	19	32	479	11.844
20	20	3	482	11.917
21	21	78	560	13.793
22	22	1	561	13.816
23	23	30	591	14.534

Figure 13: Fitted values data.

observed values and the estimated models (by default represented by circles and a solid curve, respectively).

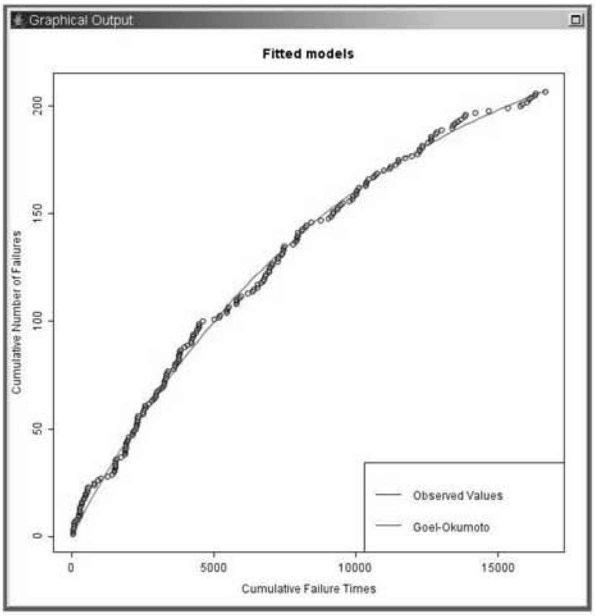


Figure 14: Fitted model plot.

5. **Plot Fitted Models**

   This option is enabled only after the analysis has been done and allows us to plot the observed values and the estimated models that we selected for the analysis.

6. **Export Output to Excel**

   This option is also enabled only after the analysis has been performed and give us the possibility to export the output of the analysis to .xls format.

3.4 Help menu

1. **Tool Help**

   Help files of the tool.

2. **SRE Help**

   Glossary of terms and background on chosen algorithms.

3. **About...**

   Information about the developers, version of the tool, etc...

4 Case study

In this section we show a demonstration of the tool using a data set by Joe (1989). The data set, reproduced in Table 1, consists of 207 observations corresponding to times between failures (ungrouped data). To load the data set in our tool click the *Data* menu and then on *Import*. An open dialog pops up (see Figure 15). We look for the data file that contains our data set (Joe_ I.xls) and then we click the *Open* button. We can distinguish three different windows in the GUI as we mentioned in the beginning of Section 3. On the left-hand side we can observe the data file while the graphical output window appears on the right-hand side. The middle of the screen is occupied by the *Select Data Type* dialog, already presented in Section 3.3 as part of the *Data Type* menu (see Figure 10). With this dialog we decide the kind of data we have. Therefore, select *Time between errors* and *Ungrouped* options. Note that in case of doubt we can try the option *Try to Autodetect Data Type*. The next step is to check whether the data presents any trend. Select the option *Trend Tests* from the *Analysis* menu and the *Trend Tests* window will be displayed (see Section 3.3, Trend Tests). Select the variable *Cumulative Times* follow by the trend tests we want to perform, in this case both Laplace and MIL-HDBK 189. Set the statistical significance level (*Alpha*) for the analysis and then click on *Perform Test*. The result of the tests is shown in Figure 11. On the *Test Results* area we can observe that both tests support the fact that there exists significant growth in reliability. In addition, the graphical output displays a graphical interpretation of the (statistical) test (in this case MIL-HDBK 189 test) as we can appreciate in Figure 16. Since the result of the trend test is positive, *i.e.*, the data shows reliability growth, the next step is to perform analysis of the data applying software reliability models. An initial model inspection can be done with the option *Model Select Wizard* from the *Analysis* menu, as we described in Section 3.3 (see Figure 9). Choose the most appropriate assumptions to our case and select the model (or models) with the highest score. In this case, Goel-Okumoto seems to be a suitable model for our assumptions, as we can appreciate in Figure 17. Select *Analyse Model*

Joe_ I data set													
39	10	4	36	4	5	4	91	49	1	25	1	4	30
42	9	49	44	32	3	78	1	30	205	5	129	103	224
186	53	14	9	2	10	1	34	170	129	4	4	35	5
5	22	36	35	121	23	33	48	32	21	4	23	9	13
165	14	22	41	12	138	95	49	62	2	35	89	99	69
22	15	19	42	14	11	41	210	16	30	37	66	9	16
14	24	12	159	89	118	29	21	18	2	114	37	46	17
1	150	382	160	66	206	9	26	62	239	13	4	85	85
240	178	34	102	9	146	59	48	25	25	111	5	31	51
6	193	27	25	96	26	30	30	17	320	78	39	13	13
19	128	34	84	40	177	349	274	82	58	31	114	39	88
84	232	108	38	86	7	22	80	239	3	39	63	152	63
80	245	196	46	152	102	9	228	220	208	78	3	83	6
212	91	3	10	172	21	173	371	40	48	126	90	149	30
317	500	673	432	66	168	66	66	128	49	332			

Table 1: Time between failures.

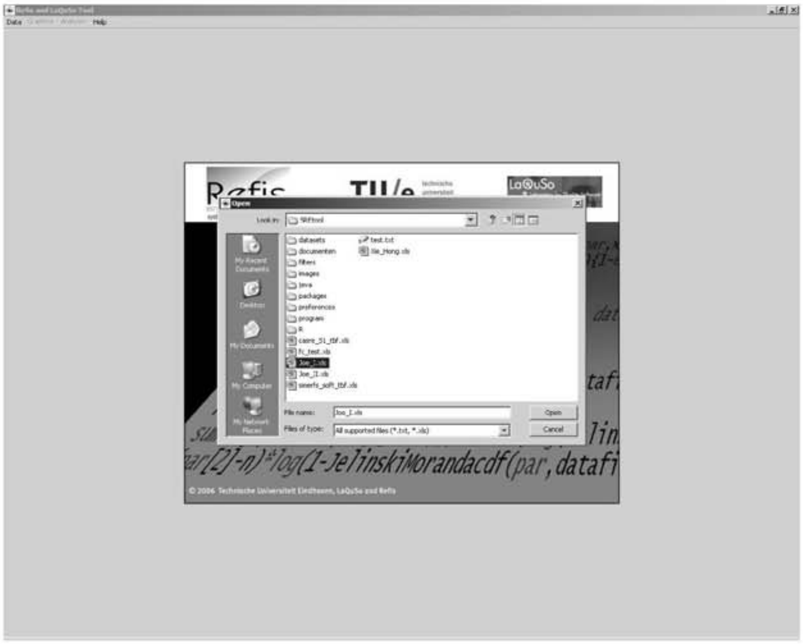


Figure 15: Browsing the data file.

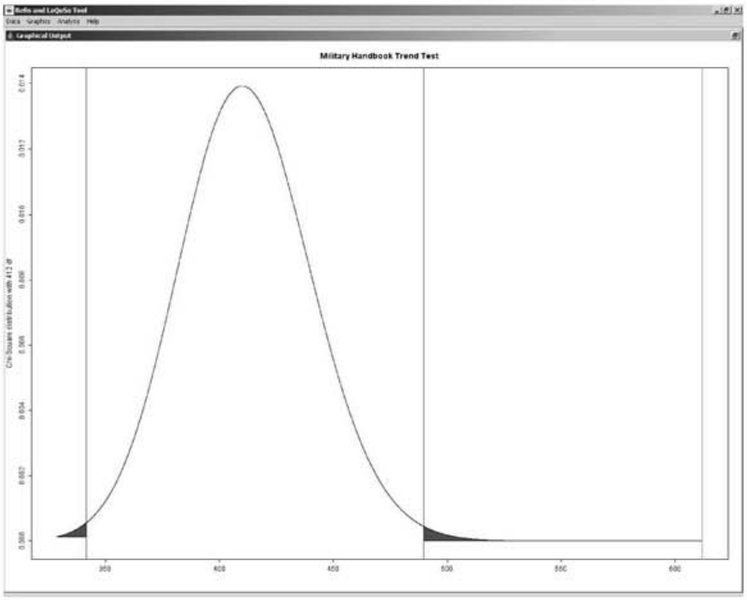
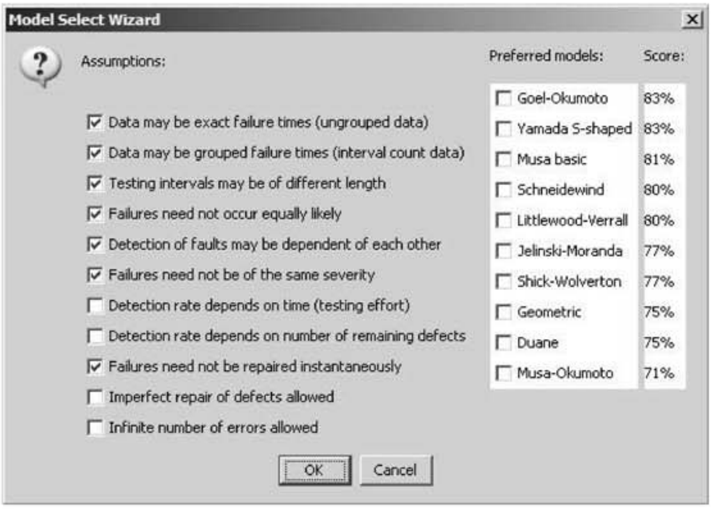


Figure 16: MIL-HDBK 189 trend test graphical interpretation.

from the *Analysis* menu and the analysis output window (see Figure 12) will be displayed. Select the Goel-Okumoto model and set the confidence level that will be used to calculate confidence intervals and the significance level to perform hypothesis testing. Choose between maximum likelihood or least squares parameter estimation methods and click on *Analyse Model*. The result of the analysis is shown in Figure 18. The data window presents the fitted values for the Goel-Okumoto model. The graphical output shows the observed values (circles)



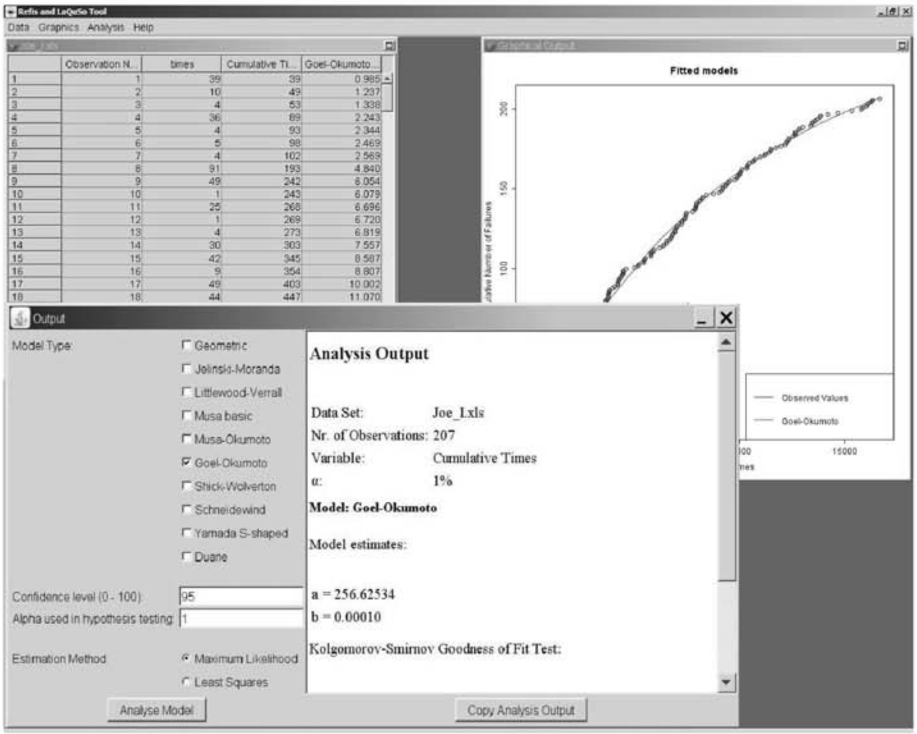
Model Select Wizard

Assumptions:

☑ Data may be exact failure times (ungrouped data)
☑ Data may be grouped failure times (interval count data)
☑ Testing intervals may be of different length
☑ Failures need not occur equally likely
☑ Detection of faults may be dependent of each other
☑ Failures need not be of the same severity
□ Detection rate depends on time (testing effort)
□ Detection rate depends on number of remaining defects
☑ Failures need not be repaired instantaneously
□ Imperfect repair of defects allowed
□ Infinite number of errors allowed

Preferred models:	Score:
□ Goel-Okumoto	83%
□ Yamada S-shaped	83%
□ Musa basic	81%
□ Schneidewind	80%
□ Littlewood-Verrall	80%
□ Jelinski-Moranda	77%
□ Shick-Wolverton	77%
□ Geometric	75%
□ Duane	75%
□ Musa-Okumoto	71%

Figure 17: After selecting our assumptions, the Goel-Okumoto model gets the highest score.



	Observation N...	times	Cumulative Ti...	Goel-Okumoto...
1	1	39	39	0.985
2	2	10	49	1.237
3	3	4	53	1.338
4	4	36	89	2.243
5	5	4	93	2.344
6	6	5	98	2.469
7	7	4	102	2.569
8	8	91	193	4.840
9	9	49	242	6.054
10	10	1	243	6.079
11	11	25	268	6.696
12	12	1	269	6.720
13	13	4	273	6.819
14	14	30	303	7.557
15	15	42	345	8.587
16	16	9	354	8.807
17	17	49	403	10.002
18	18	44	447	11.070

Output

Model Type: □ Geometric □ Jelinski-Moranda □ Littlewood-Verrall □ Musa basic □ Musa-Okumoto ☑ Goel-Okumoto □ Shick-Wolverton □ Schneidewind □ Yamada S-shaped □ Duane

Confidence level (0 - 100): 95
Alpha used in hypothesis testing: 1

Estimation Method ● Maximum Likelihood ○ Least Squares

Analysis Output

Data Set: Joe_I.xls
Nr. of Observations: 207
Variable: Cumulative Times
α: 1%

Model: Goel-Okumoto

Model estimates:

a = 256.62534
b = 0.00010

Kolgomorov-Smirnov Goodness of Fit Test:

Figure 18: Model analysis.

and the estimated model (curve). A first inspection of this plot suggests that Goel-Okumoto is a reasonable model for our data. Formal result of the analysis (parameter estimates and goodness-of-fit test) is displayed on the analysis output window. As a consequence of our analysis we can conclude that there is no evidence to reject the fact that our data can be described using the Goel-Okumoto model.

5 Conclusions and future work

We have developed a new tool for statistical software reliability analyses. The interface is programmed in Java and therefore, it is platform independent. Our tool uses well-documented state-of-the-art statistical algorithms programmed in R and encourages to apply best practices from statistics. Furthermore, it has a user-friendly interface which includes features like model selection wizard and auto detection of data type. The extension of the tool incorporating new models and features is an ongoing challenge that stimulates us to continue working on this direction.

References

M. Bhattacharjee, J. V. Deshpande, and U. V. Naik-Nimbalkar. Unconditional tests of goodness of fit for the intensity of time-truncated nonhomogeneous Poisson processes. *Technometrics*, 46(3):330–338, 2004.

A.L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Trans. Soft. Eng.*, 11(12):1411–1423, 1985.

H. Joe. Statistical inference for General-Order-Statistics and Nonhomogeneous-Poisson-Process software reliability models. *IEEE Trans. Software Eng.*, 15(11):1485–1490, 1989.

V.S. Kharchenko, O.M. Tarasyuk, V.V. Sklyar, and V.Yu. Dubnitsky. The method of software reliability growth models choice using assumptions matrix. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 541–546, Washington, DC, USA, 2002. IEEE Computer Society.

M.R. Lyu, editor. *Handbook of Software Reliability Engineering.* McGraw-Hill and IEEE Computer Society, New York, 1996.

J.D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper.* Author House, Bloomington, USA, 2nd edition, 2006.

M. Ohba. Software reliability analysis models. *IBM J. Res. Develop.*, 28(4):428–443, 1984.

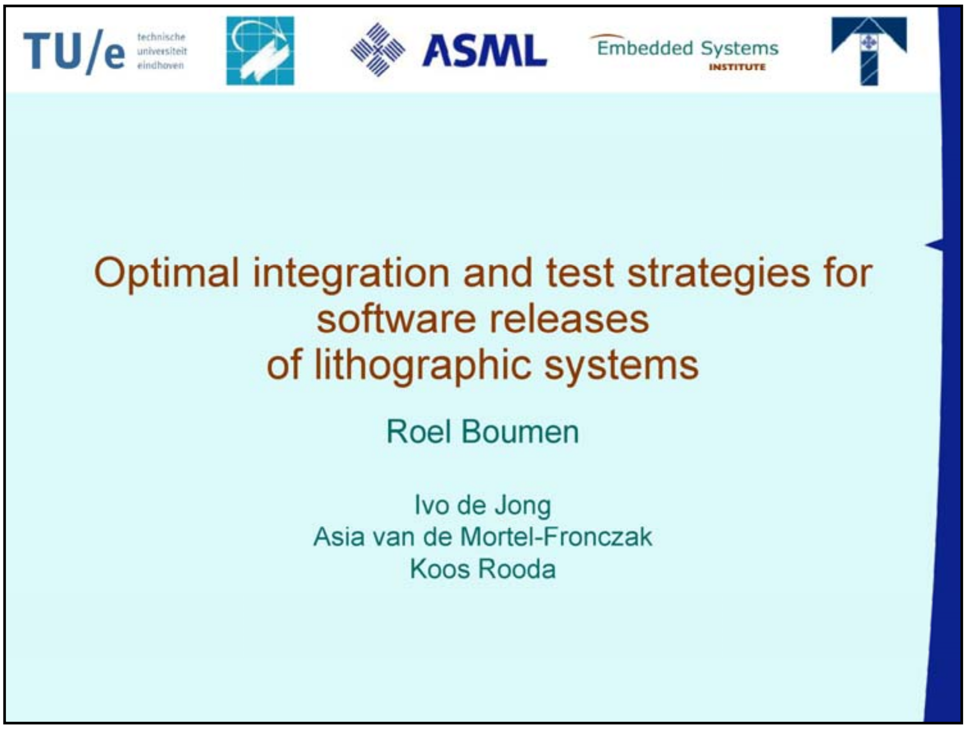
H. Pham. *System Software Reliability.* Springer Series in Reliability Engineering. Springer, London, 2006.

S.E. Rigdon and A.P. Basu. *Statistical Methods for the Reliability of Repairable Systems.* Wiley, 2000.

M. Xie, Y.-S. Dai, and K.-L. Poh. *Computing Systems Reliability. Models and Analysis.* Kluwer, New York, 2004.

L. Yin and K.S. Trivedi. Confidence interval estimation of NHPP-based software reliability models. In *Proc. 10th Int. Symp. Software Reliability Engineering (ISSRE 1999)*, pages 6–11, 1999.

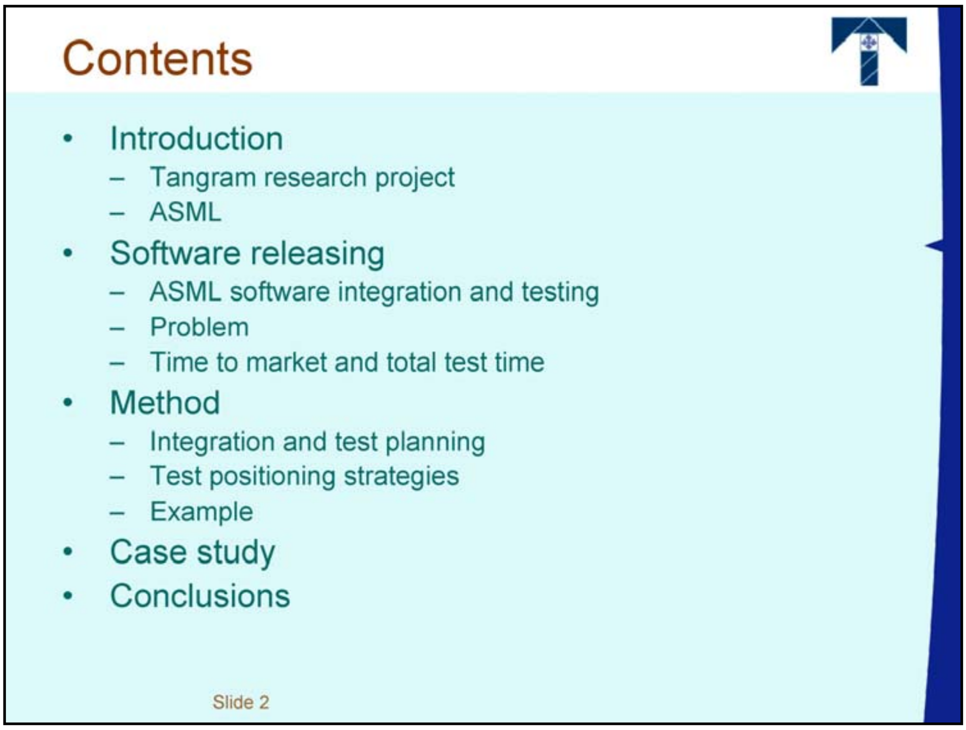
J. Zhao and J. Wang. A new goodness-of-fit test based on the Laplace statistic for a large class of NHPP models. *Comm. Statist. Simulation Comput.*, 34(3):725–736, 2005.



Optimal integration and test strategies for software releases of lithographic systems

Roel Boumen

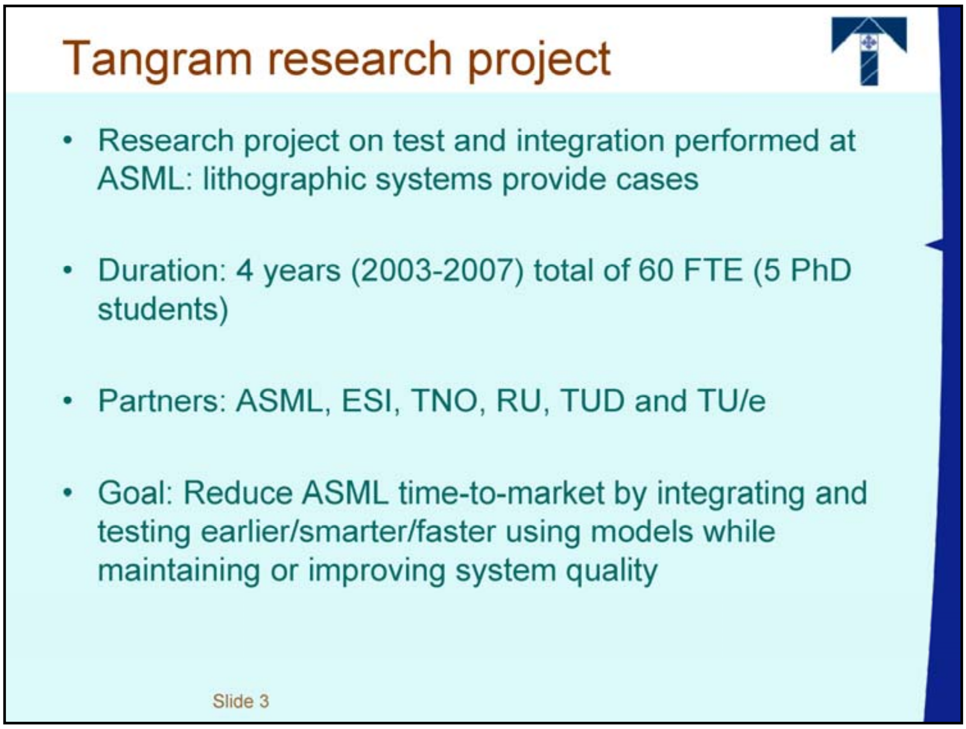
Ivo de Jong
Asia van de Mortel-Fronczak
Koos Rooda



Contents

- Introduction
  - Tangram research project
  - ASML
- Software releasing
  - ASML software integration and testing
  - Problem
  - Time to market and total test time
- Method
  - Integration and test planning
  - Test positioning strategies
  - Example
- Case study
- Conclusions

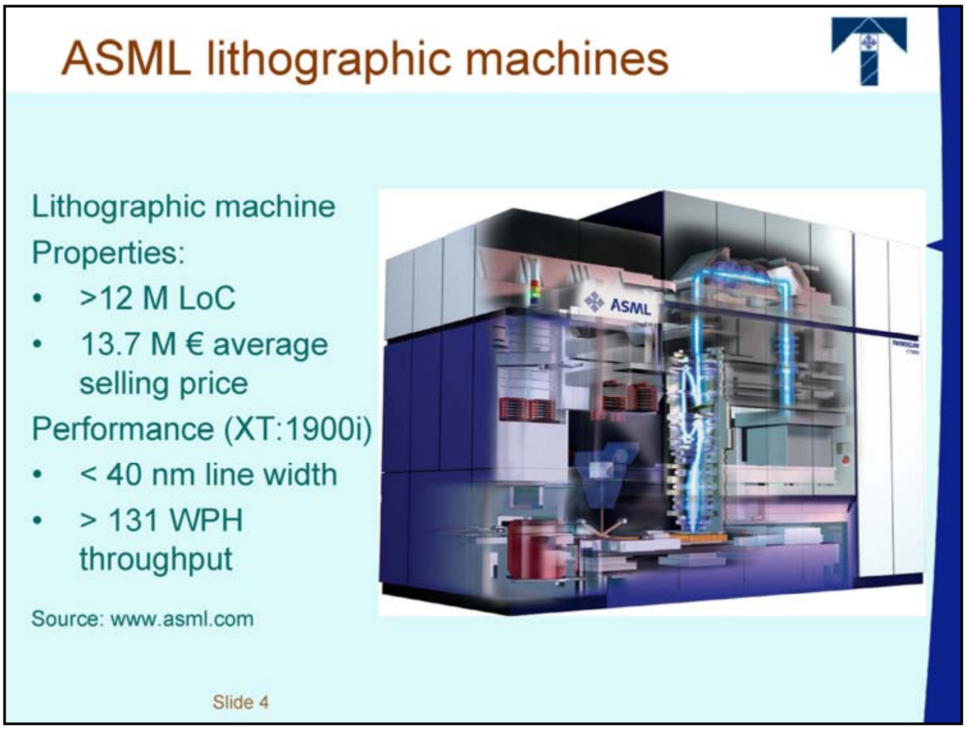
Slide 2



Tangram research project

- Research project on test and integration performed at ASML: lithographic systems provide cases
- Duration: 4 years (2003-2007) total of 60 FTE (5 PhD students)
- Partners: ASML, ESI, TNO, RU, TUD and TU/e
- Goal: Reduce ASML time-to-market by integrating and testing earlier/smarter/faster using models while maintaining or improving system quality

Slide 3



ASML lithographic machines

Lithographic machine

Properties:

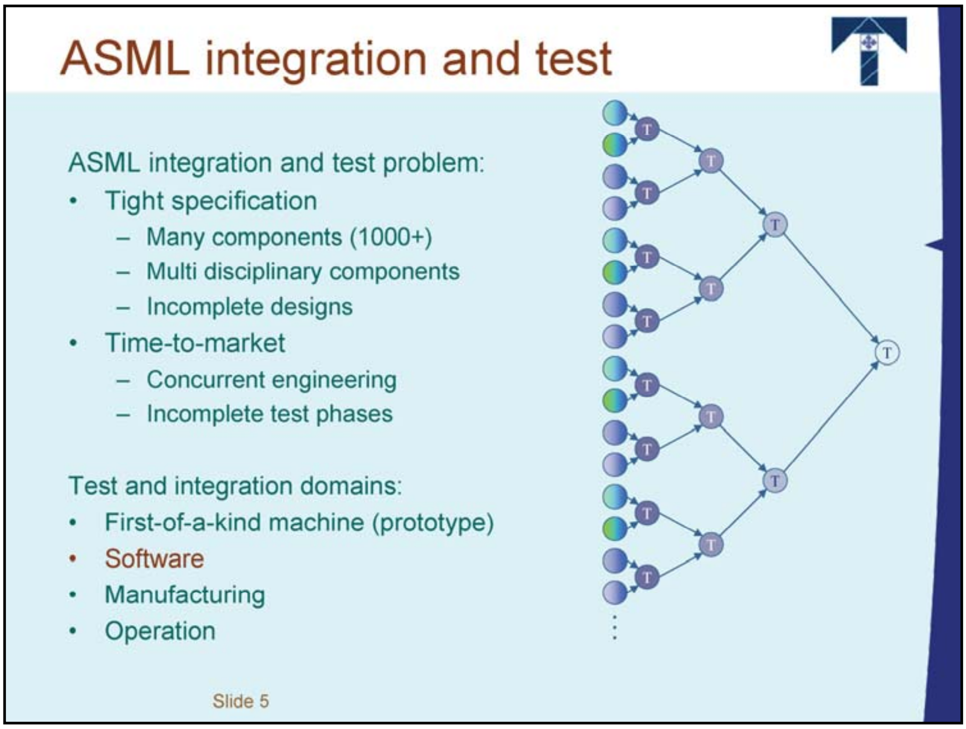
- >12 M LoC
- 13.7 M € average selling price

Performance (XT:1900i)

- < 40 nm line width
- > 131 WPH throughput

Source: www.asml.com

Slide 4



ASML integration and test

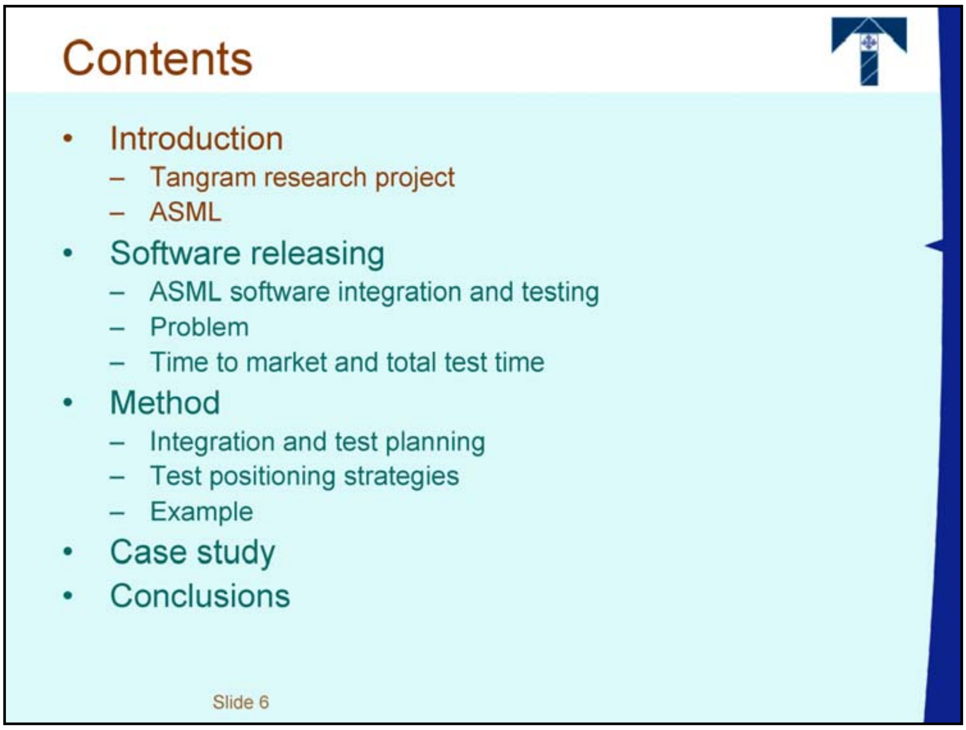
ASML integration and test problem:

- Tight specification
  - Many components (1000+)
  - Multi disciplinary components
  - Incomplete designs
- Time-to-market
  - Concurrent engineering
  - Incomplete test phases

Test and integration domains:

- First-of-a-kind machine (prototype)
- Software
- Manufacturing
- Operation

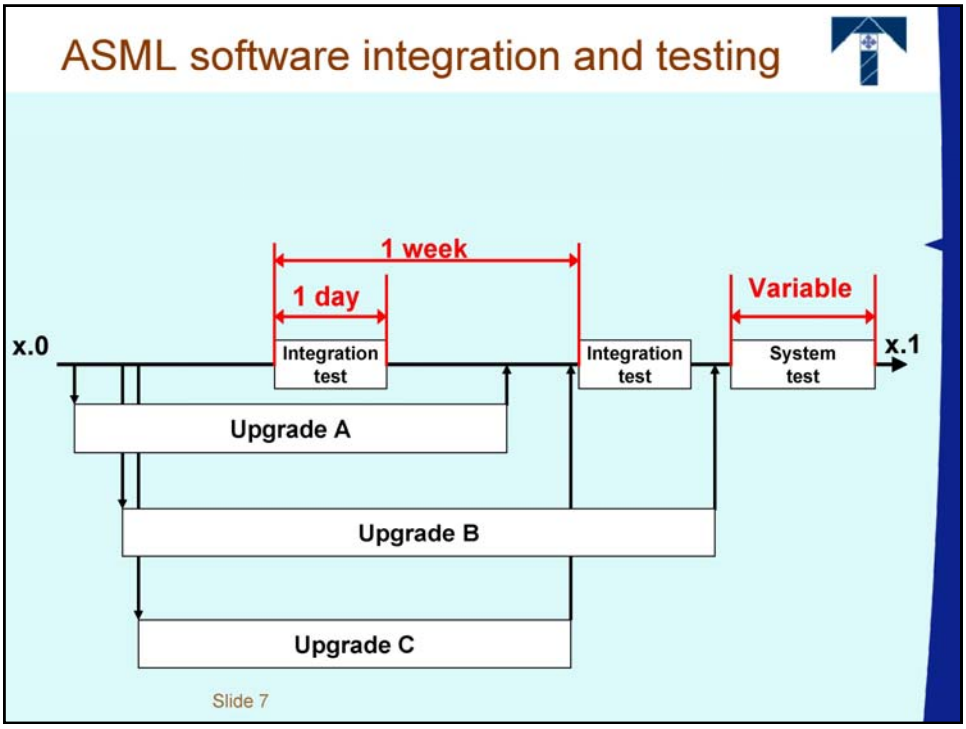
Slide 5



Contents

- Introduction
  - Tangram research project
  - ASML
- Software releasing
  - ASML software integration and testing
  - Problem
  - Time to market and total test time
- Method
  - Integration and test planning
  - Test positioning strategies
  - Example
- Case study
- Conclusions

Slide 6



ASML software integration and testing

x.0

1 week

1 day

Integration test

Integration test

Variable

System test

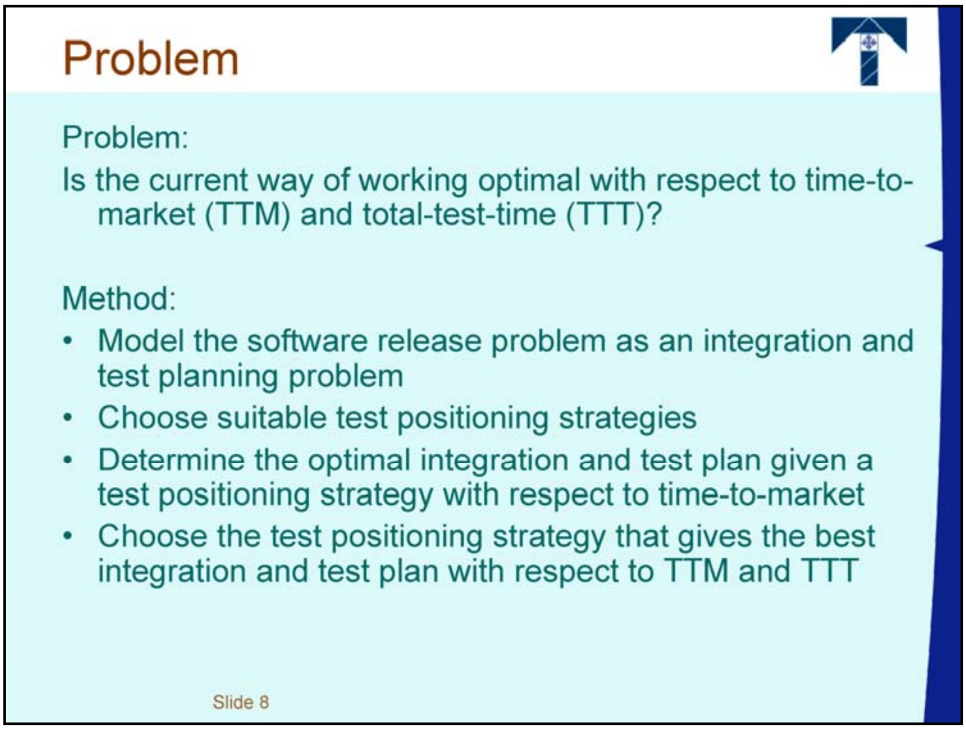
x.1

Upgrade A

Upgrade B

Upgrade C

Slide 7



Problem

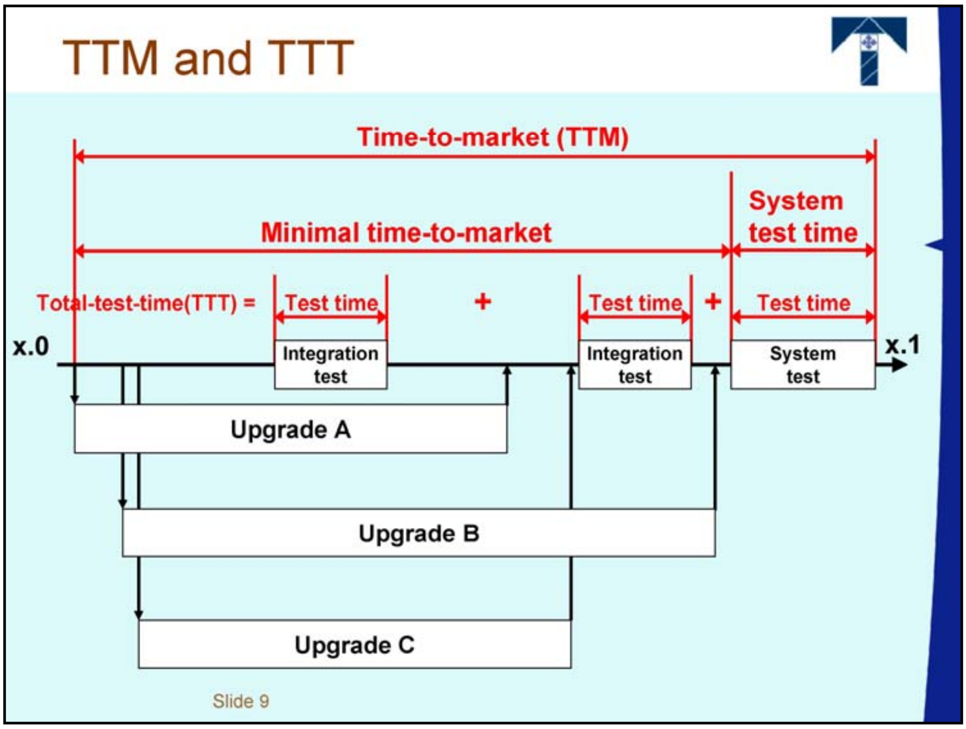
Problem:

Is the current way of working optimal with respect to time-to-market (TTM) and total-test-time (TTT)?

Method:

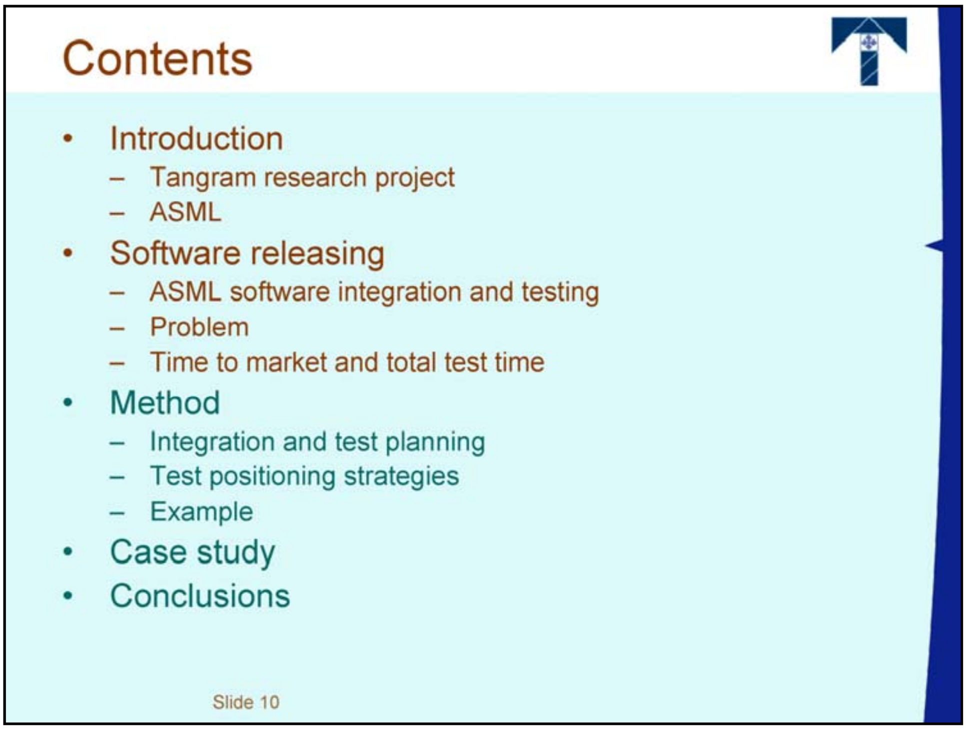
- Model the software release problem as an integration and test planning problem
- Choose suitable test positioning strategies
- Determine the optimal integration and test plan given a test positioning strategy with respect to time-to-market
- Choose the test positioning strategy that gives the best integration and test plan with respect to TTM and TTT

Slide 8



TTM and TTT

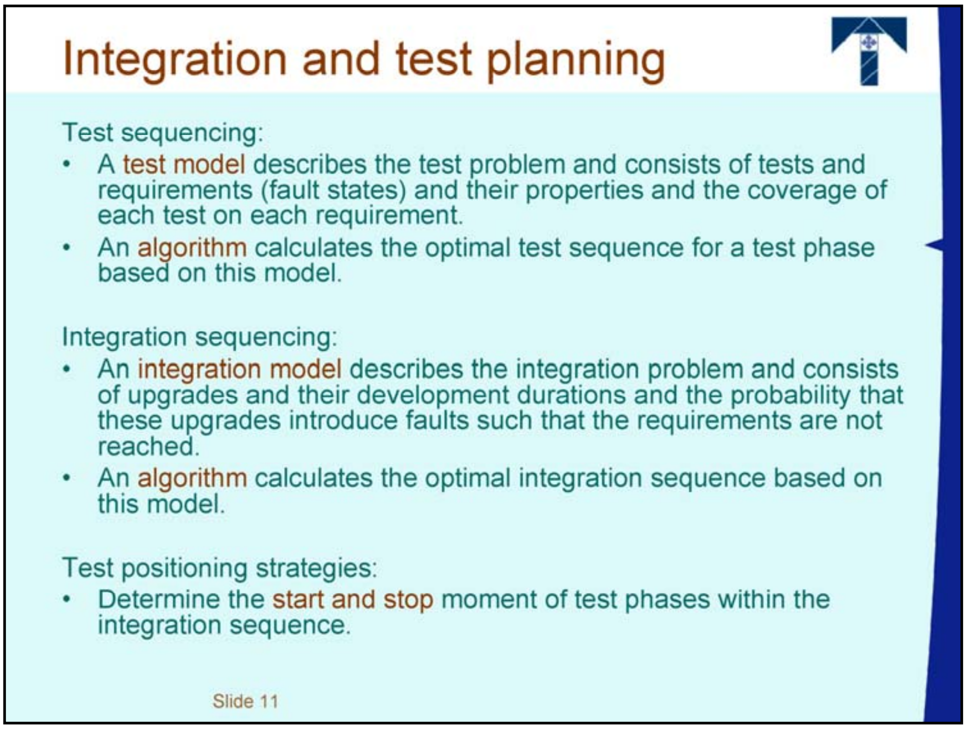
Slide 9



Contents

- Introduction
  - Tangram research project
  - ASML
- Software releasing
  - ASML software integration and testing
  - Problem
  - Time to market and total test time
- Method
  - Integration and test planning
  - Test positioning strategies
  - Example
- Case study
- Conclusions

Slide 10



Integration and test planning

Test sequencing:

- A test model describes the test problem and consists of tests and requirements (fault states) and their properties and the coverage of each test on each requirement.
- An algorithm calculates the optimal test sequence for a test phase based on this model.

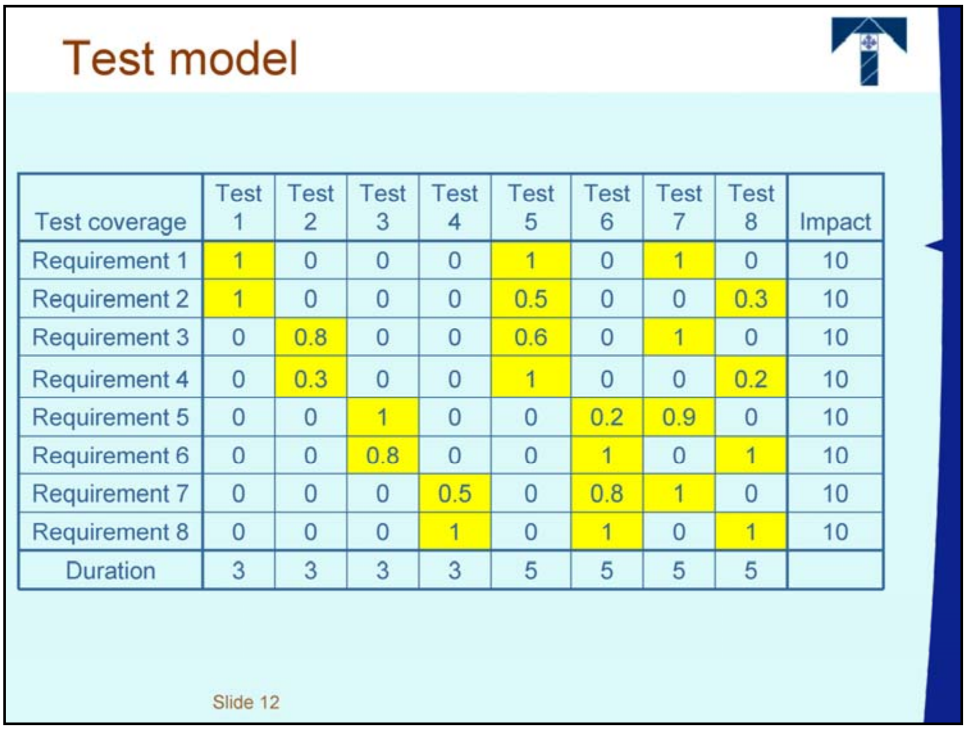
Integration sequencing:

- An integration model describes the integration problem and consists of upgrades and their development durations and the probability that these upgrades introduce faults such that the requirements are not reached.
- An algorithm calculates the optimal integration sequence based on this model.

Test positioning strategies:

- Determine the start and stop moment of test phases within the integration sequence.

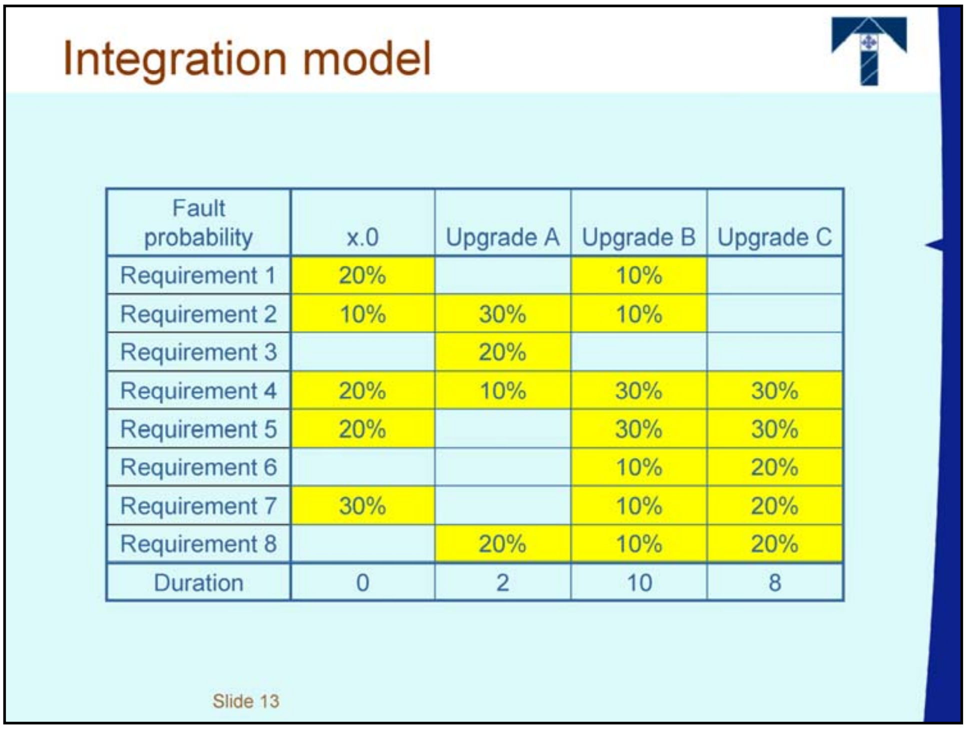
Slide 11



Test model

Test coverage	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Impact
Requirement 1	1	0	0	0	1	0	1	0	10
Requirement 2	1	0	0	0	0.5	0	0	0.3	10
Requirement 3	0	0.8	0	0	0.6	0	1	0	10
Requirement 4	0	0.3	0	0	1	0	0	0.2	10
Requirement 5	0	0	1	0	0	0.2	0.9	0	10
Requirement 6	0	0	0.8	0	0	1	0	1	10
Requirement 7	0	0	0	0.5	0	0.8	1	0	10
Requirement 8	0	0	0	1	0	1	0	1	10
Duration	3	3	3	3	5	5	5	5	

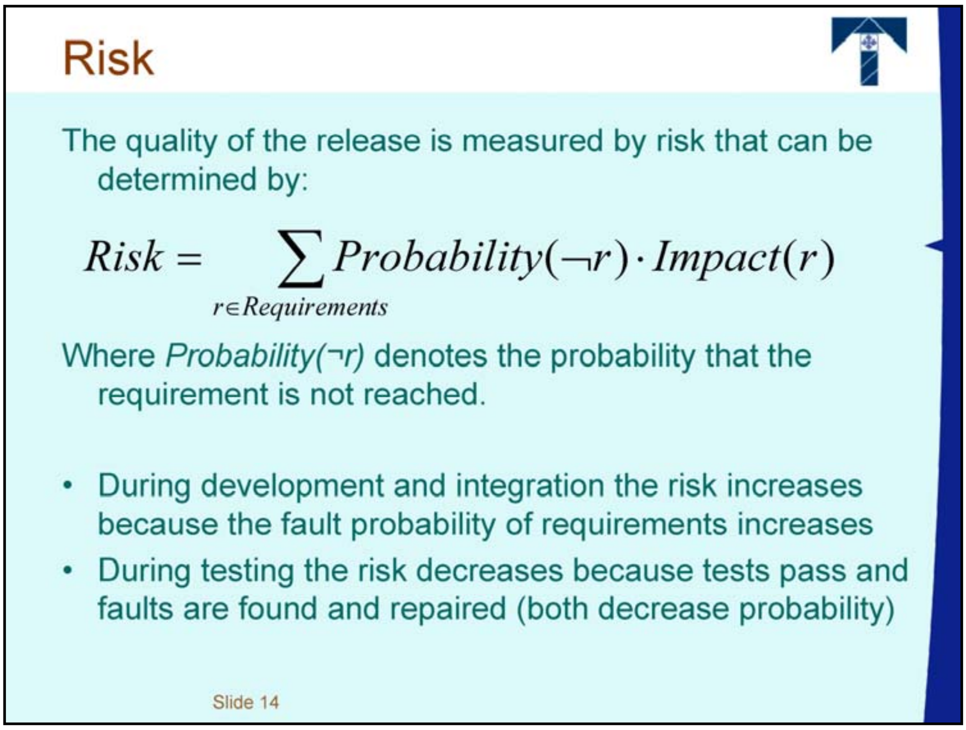
Slide 12



Integration model

Fault probability	x.0	Upgrade A	Upgrade B	Upgrade C
Requirement 1	20%		10%	
Requirement 2	10%	30%	10%	
Requirement 3		20%		
Requirement 4	20%	10%	30%	30%
Requirement 5	20%		30%	30%
Requirement 6			10%	20%
Requirement 7	30%		10%	20%
Requirement 8		20%	10%	20%
Duration	0	2	10	8

Slide 13



Risk

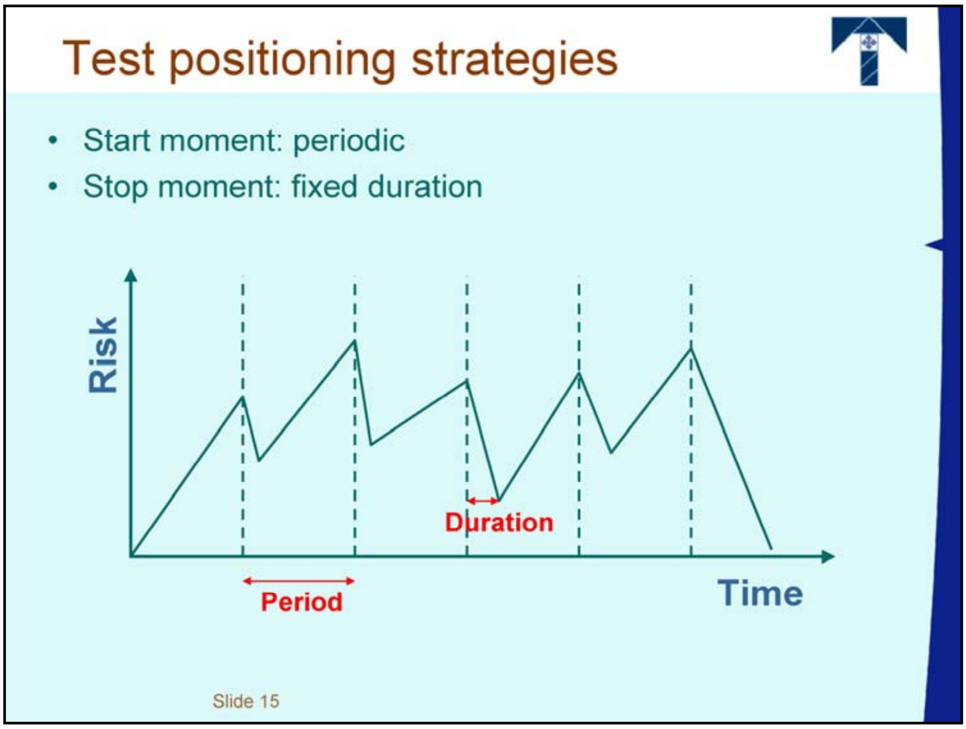
The quality of the release is measured by risk that can be determined by:

$$Risk = \sum_{r \in Requirements} Probability(\neg r) \cdot Impact(r)$$

Where *Probability(¬r)* denotes the probability that the requirement is not reached.

- During development and integration the risk increases because the fault probability of requirements increases
- During testing the risk decreases because tests pass and faults are found and repaired (both decrease probability)

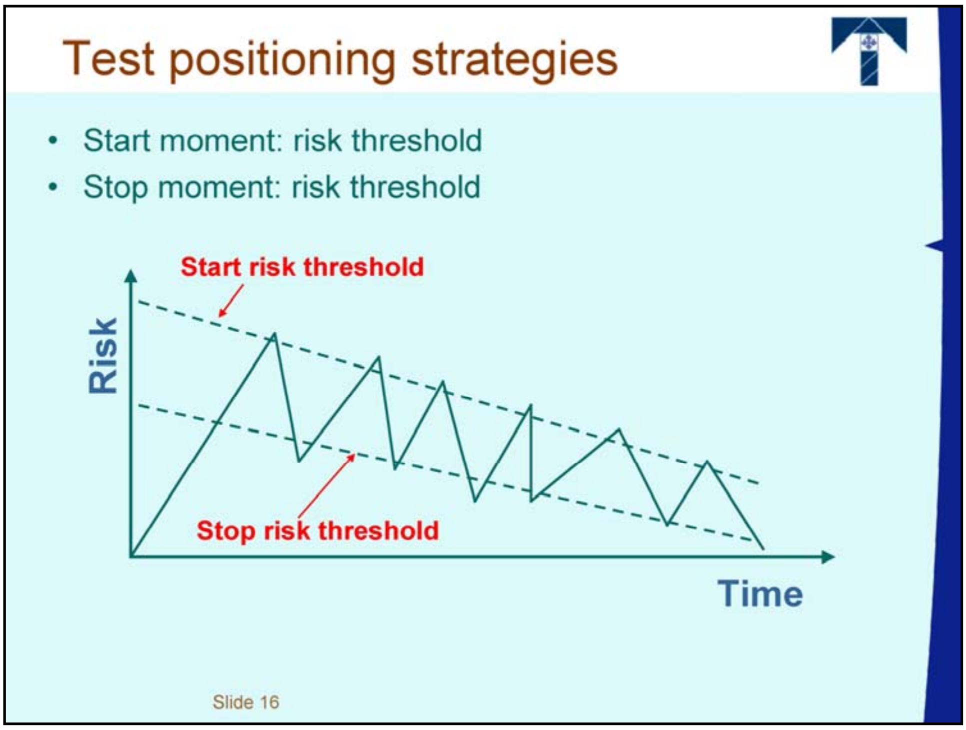
Slide 14



Test positioning strategies

- Start moment: periodic
- Stop moment: fixed duration

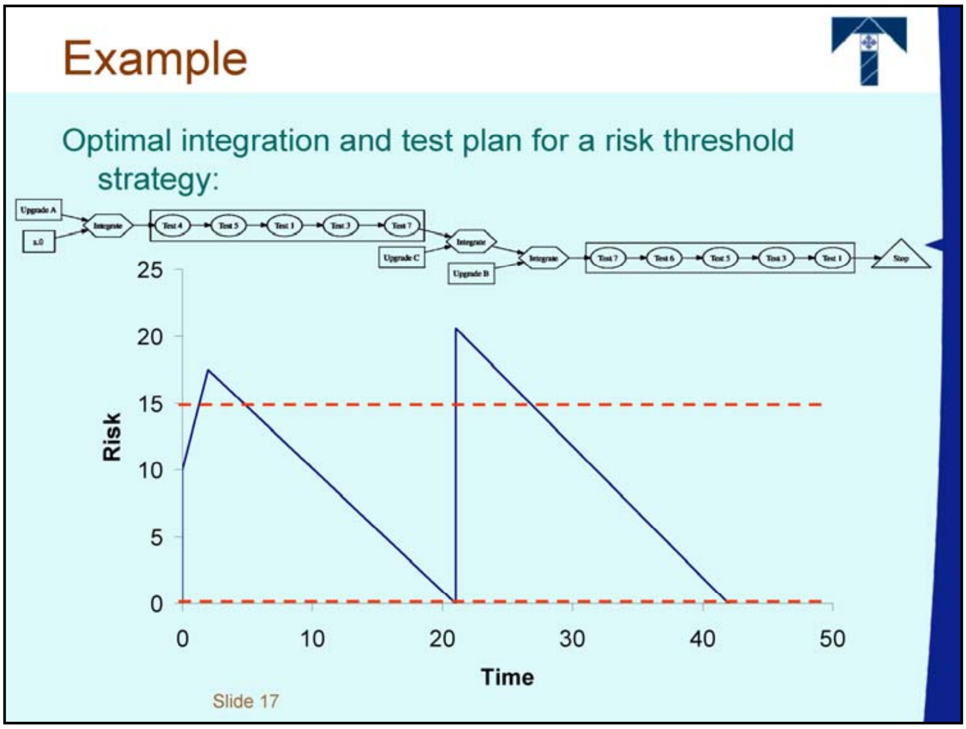
Slide 15



Test positioning strategies

- Start moment: risk threshold
- Stop moment: risk threshold

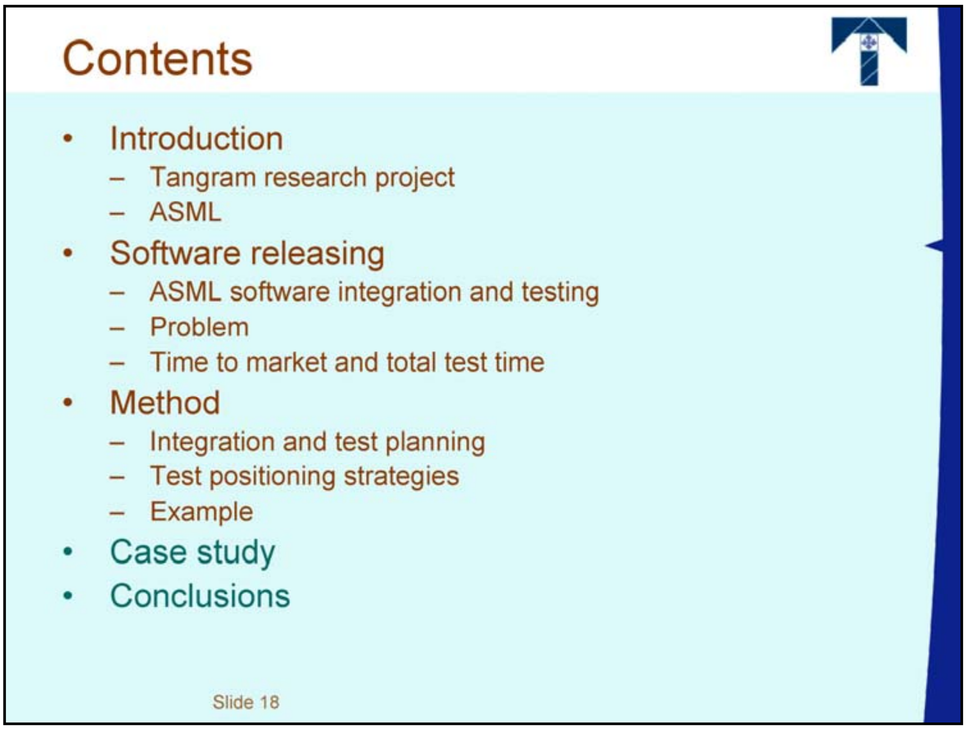
Slide 16



Example

Optimal integration and test plan for a risk threshold strategy:

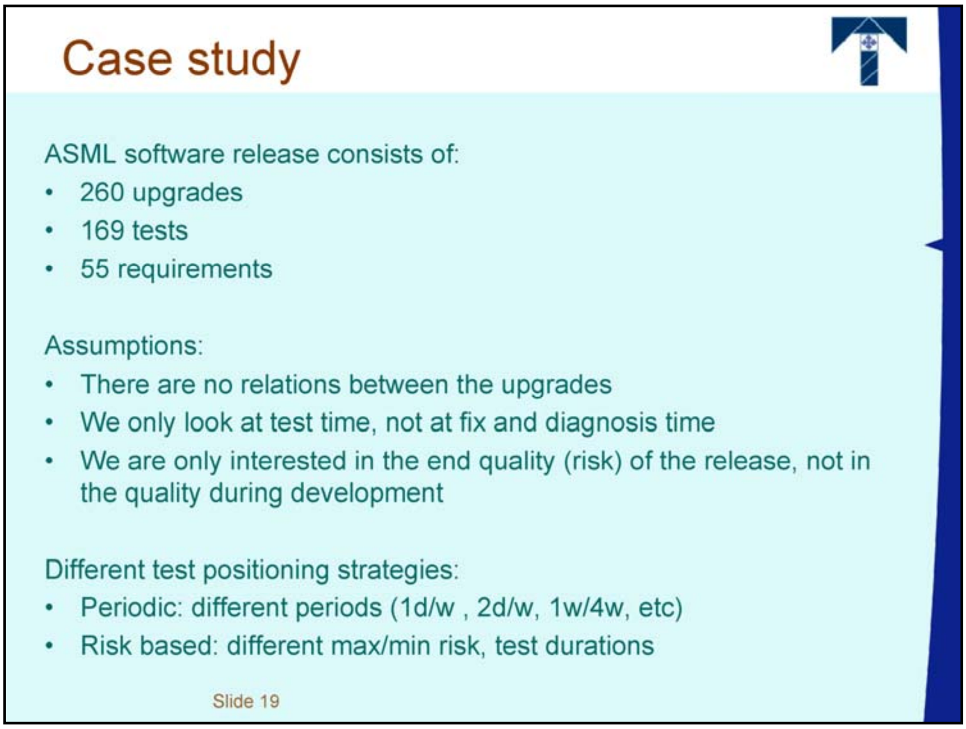
Slide 17



Contents

- Introduction
  - Tangram research project
  - ASML
- Software releasing
  - ASML software integration and testing
  - Problem
  - Time to market and total test time
- Method
  - Integration and test planning
  - Test positioning strategies
  - Example
- Case study
- Conclusions

Slide 18



Case study

ASML software release consists of:

- 260 upgrades
- 169 tests
- 55 requirements

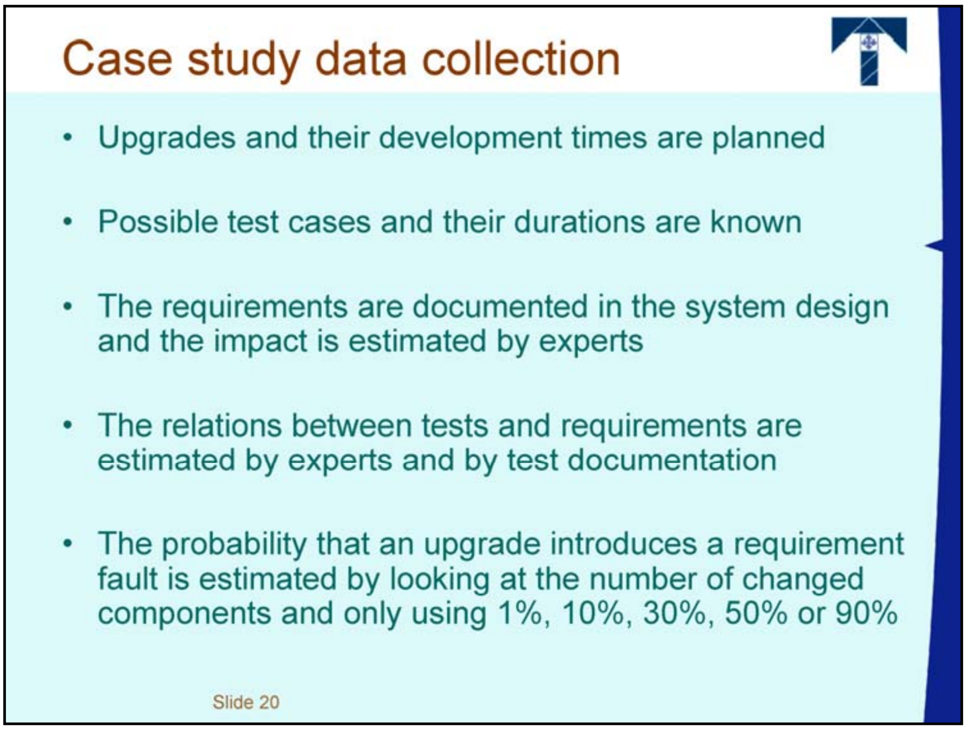
Assumptions:

- There are no relations between the upgrades
- We only look at test time, not at fix and diagnosis time
- We are only interested in the end quality (risk) of the release, not in the quality during development

Different test positioning strategies:

- Periodic: different periods (1d/w , 2d/w, 1w/4w, etc)
- Risk based: different max/min risk, test durations

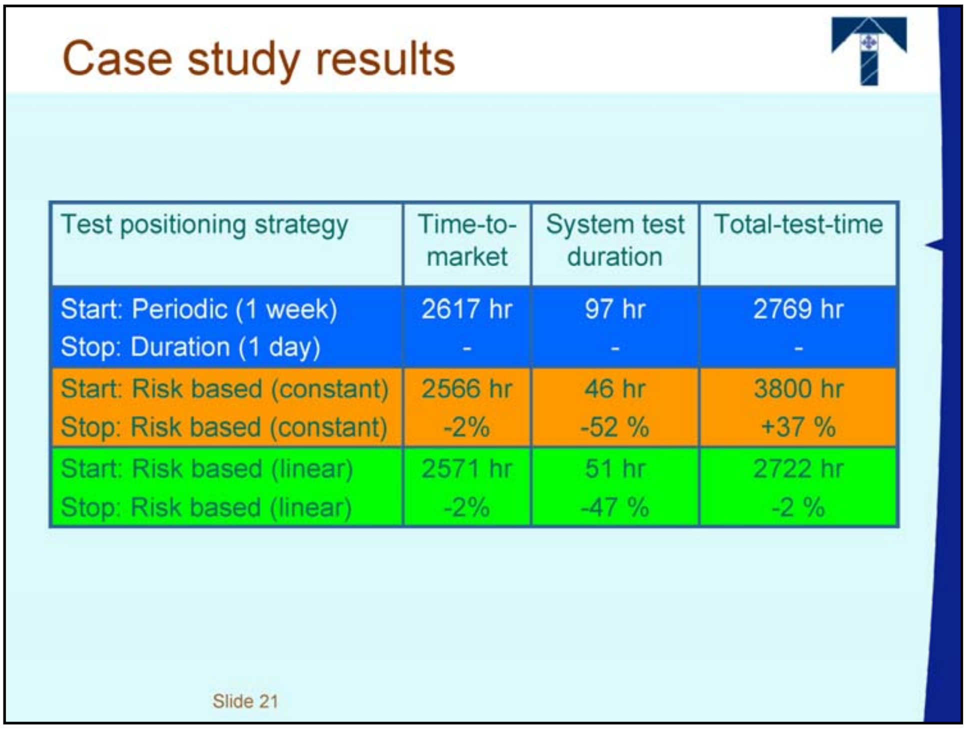
Slide 19



Case study data collection

- Upgrades and their development times are planned
- Possible test cases and their durations are known
- The requirements are documented in the system design and the impact is estimated by experts
- The relations between tests and requirements are estimated by experts and by test documentation
- The probability that an upgrade introduces a requirement fault is estimated by looking at the number of changed components and only using 1%, 10%, 30%, 50% or 90%

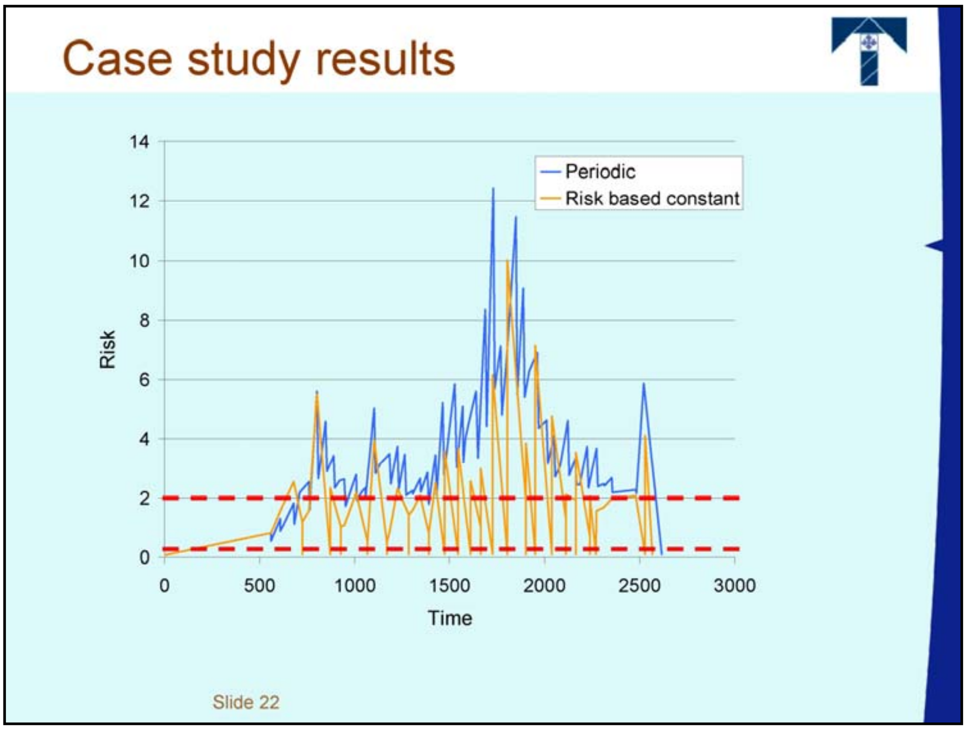
Slide 20



Case study results

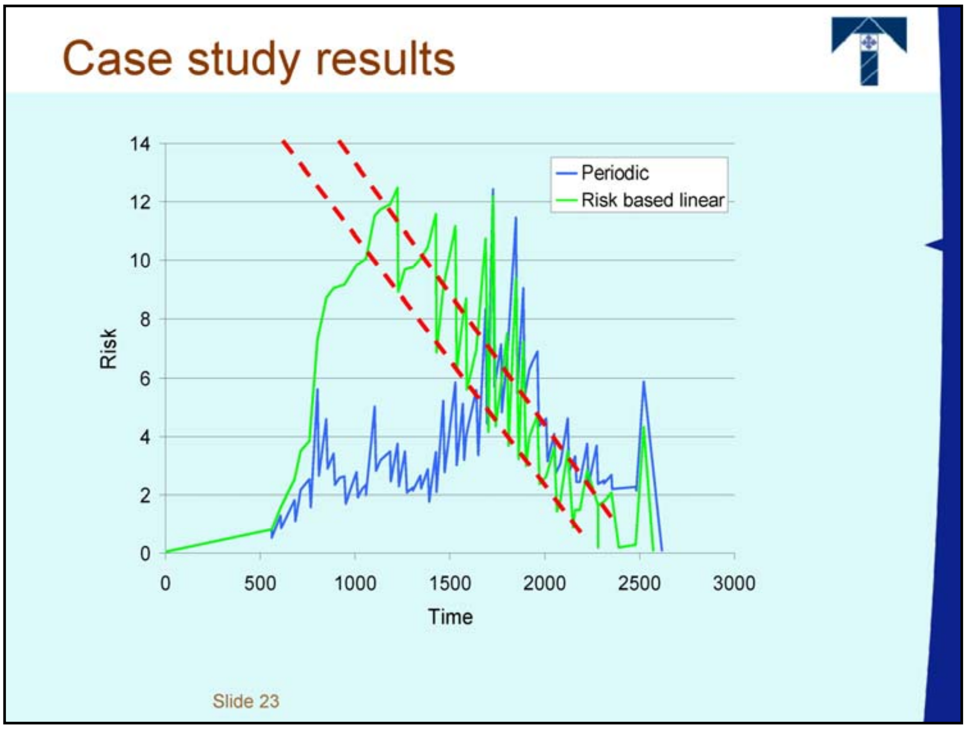
Test positioning strategy	Time-to-market	System test duration	Total-test-time
Start: Periodic (1 week) Stop: Duration (1 day)	2617 hr -	97 hr -	2769 hr -
Start: Risk based (constant) Stop: Risk based (constant)	2566 hr -2%	46 hr -52 %	3800 hr +37 %
Start: Risk based (linear) Stop: Risk based (linear)	2571 hr -2%	51 hr -47 %	2722 hr -2 %

Slide 21



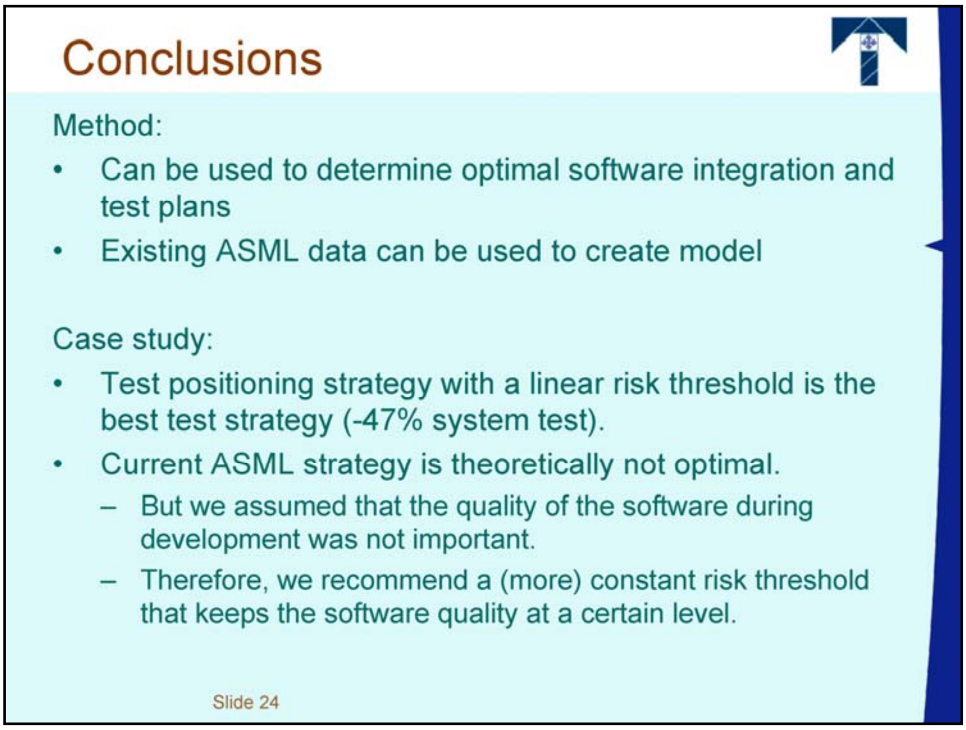
Case study results

Slide 22



Case study results

Slide 23



Conclusions

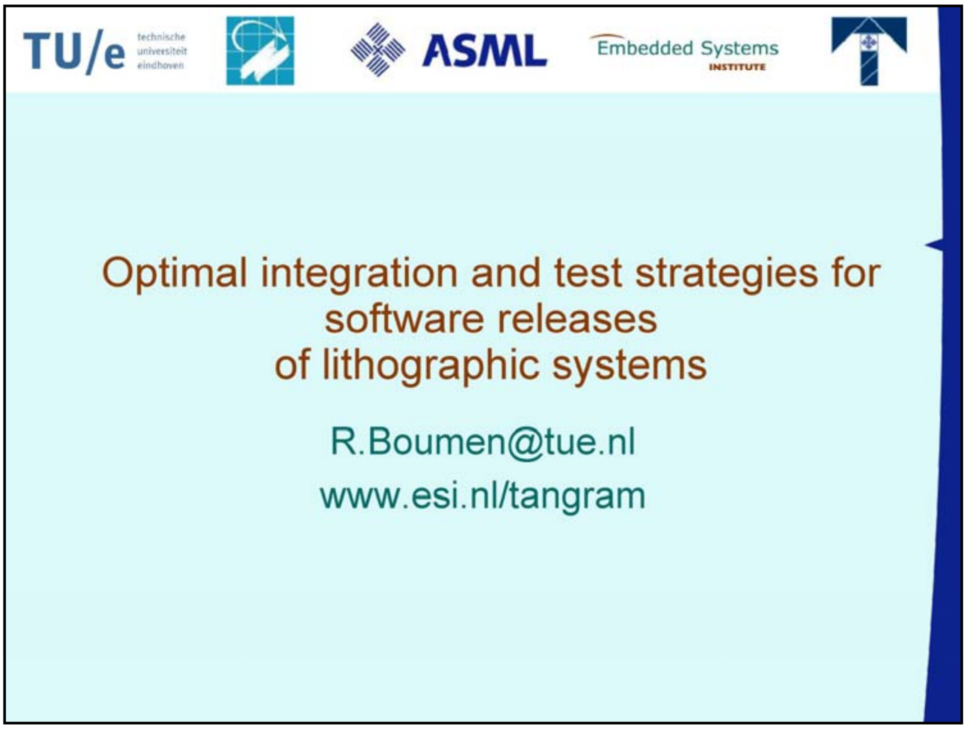
Method:

- Can be used to determine optimal software integration and test plans
- Existing ASML data can be used to create model

Case study:

- Test positioning strategy with a linear risk threshold is the best test strategy (-47% system test).
- Current ASML strategy is theoretically not optimal.
  - But we assumed that the quality of the software during development was not important.
  - Therefore, we recommend a (more) constant risk threshold that keeps the software quality at a certain level.

Slide 24



TU/e technische universiteit eindhoven
ASML
Embedded Systems INSTITUTE

Optimal integration and test strategies for software releases of lithographic systems

R.Boumen@tue.nl

www.esi.nl/tangram

Static Memory and Timing Analysis of Embedded Systems Code

Christian Ferdinand *Reinhold Heckmann* *Bärbel Franzen*

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany

Phone: +49-681-38360-0 e-mail: info@absint.com
Fax: +49-681-38360-20 Web page: http://www.absint.com

Abstract

Failure of a safety-critical application on an embedded processor can lead to severe damage or even loss of life. Here we are concerned with two kinds of failure: stack overflow, which usually leads to runtime errors that are difficult to diagnose, and failure to meet deadlines, which is catastrophic for systems with hard real-time characteristics. Classical validation methods like code review and testing with repeated measurements require a lot of effort, are expensive, and do not really help in proving the absence of such errors. **AbsInt**'s tools **StackAnalyzer** and **aiT** (timing analyzer) provide a solution to this problem. They use abstract interpretation as a formal method that allows to obtain statements valid for all program runs with all inputs.

1 Introduction

The use of safety-critical embedded software in the automotive and avionics industries is increasing rapidly. Failure of such a safety-critical embedded system may result in the loss of life or in large damages. Also for non-safety-critical applications, software failure may necessitate expensive updates. Therefore, utmost carefulness and state-of-the-art machinery have to be applied to make sure that an application meets all requirements. To do so lies in the responsibility of the system designer(s).

Traditional certification standards evaluate the quality of a software system by assessing the quality of the development process that produced it. Yet the benefit of such a process-based assurance is limited. Therefore, switching to a product-based assurance process is advised, which judges the quality of a software product by examining its properties.

Classical software validation methods like code review and testing with debugging are very expensive and cannot really guarantee the absence of errors. Formal verification methods provide an alternative, in particular for safety-critical applications. One such method is *abstract interpretation* [2], which allows to obtain statements that are valid for all program runs with all inputs. Such statements may be absence of violations of timing or space constraints, or absence of runtime errors. For example, stack overflow can be detected by **AbsInt**'s **StackAnalyzer**, and violations of timing constraints are found by **AbsInt**'s **aiT** tool [5] that determines upper bounds for the worst-case execution times of the tasks of an application. Among other things, these tools perform a *value analysis* as the principal source of information about the values manipulated by the analyzed program.

2 Value Analysis

Value analysis tries to determine the values stored in the processor's memory for every program point and execution context. Often, it is sufficient to restrict value analysis to the processor registers, but sometimes, it is useful to get information about main memory as well.

Value analysis is a static analysis method producing results valid for every program run and all inputs to the program. Therefore, it cannot always predict an exact value for a memory location, but determines *abstract values* instead that stand for sets of concrete values. More precisely, it computes for each program point and execution context an *abstract state* that maps memory locations to abstract values. Each machine instruction is modeled by a transfer function mapping input states to output states in a way that is compatible with the semantics of the instruction. At control-flow joins, the incoming abstract states are combined into a single outgoing state using a combination function. Because of the presence of loops, transfer and combination functions must be applied repeatedly until the system of abstract states stabilizes. Termination of this fixed-point iteration is ensured on a theoretical level by the monotonicity of transfer and combination functions and the fact that a memory location can only hold finitely many different values. Practically, value analysis becomes only efficient by applying suitable widening and narrowing operators as proposed in [2].

There are several variants of value analysis depending on what kinds of abstract values are used. The simplest form of value analysis is *constant propagation*: an abstract value is either a single concrete value or the statement that no information about the value is known. A more elaborate form of value analysis computes safe lower and upper bounds for the possible concrete values, i.e. abstract values are intervals that are guaranteed to contain the exact values.

All the value analysis variants described above determine sets of possible values for individual memory locations without any relationship between these sets. Yet if an unknown value is moved from register `r1` to `r2`, then both registers contain unknown values afterward, but these values are known to be equal. So a possible extension of value analysis may record known equalities between values, or more generally, upper and lower bounds for their differences, or even more generally, arbitrary linear constraints between values.

Value analysis, even in its simple form as interval analysis, has various applications as an auxiliary method providing input for other analysis tasks. Some of these applications are listed in the sequel.

3 Stack Usage Analysis

A possible cause of catastrophic failure is stack overflow that usually leads to run-time errors that are difficult to diagnose. The problem is that the memory area for the stack usually must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for a single program run with fixed input. Even repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed. Some, but not all compilers provide information about stack usage, but this requires the availability of the source code, and the information becomes invalid when the generated code is optimized by hand or by some automatic tool.

AbsInt's tool **StackAnalyzer** provides a solution to this problem: By concentrating on the value of the stack pointer during value analysis, the tool can figure out how the stack increases and decreases along the various control-flow paths. This information can be used to derive the maximum stack usage of the entire task.

The results of **StackAnalyzer** are presented as annotations in a combined call graph and control-flow graph. Figure 1 shows the call graph of a small application, with stack analysis results at routines and for the entire application (at the top). On this level, the results of stack analysis are displayed in boxes located to the right of the boxes representing the routines of the application. Each result box carries two results: a *global result*, coming first, and a *local result*, following in angular brackets. Each result is an interval of possible stack levels. Intervals of the form $[n, n]$ are abbreviated to n.

The local result at a routine R indicates the stack usage in R considered on its own: It is an interval showing the possible range of stack levels within the routine, assuming value 0 at routine entry. The local result for a routine is derived from the results at individual instructions, which are shown in Figure 2 for one of the routines of this example.

The global result for routine R indicates the stack usage of R in the context of the entire application. It is an interval providing bounds for the stack level while the processor is executing instructions of R, for all call

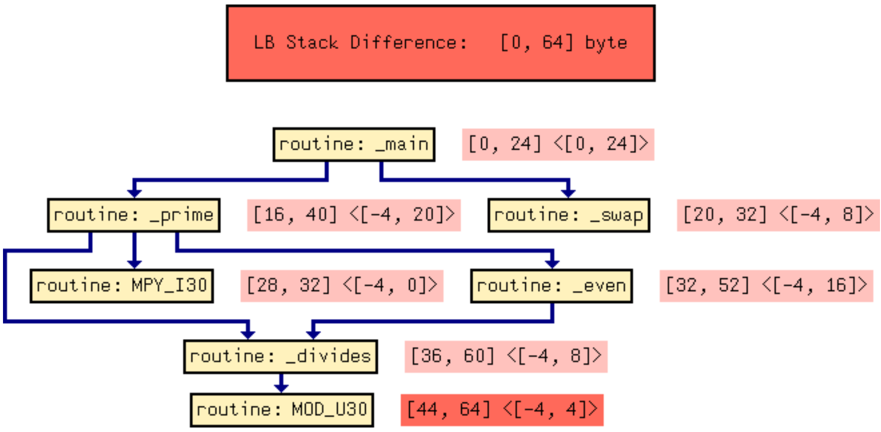


Figure 1: Call graph with stack analysis results

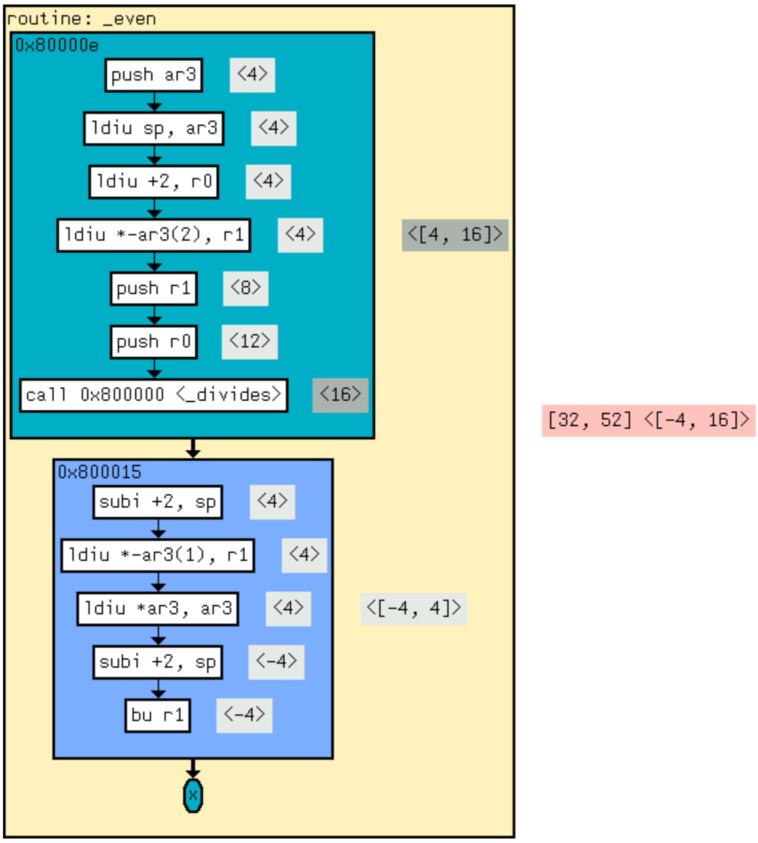


Figure 2: Individual instructions with stack analysis results

paths from the entry point to *R*. Thus, the global result at routine *R* does not include the stack usage of the routines called by *R*.

StackAnalyzer provides automatic tool support to calculate precise information on the stack usage. This not only reduces development effort, but also helps to prevent runtime errors due to stack overflow. Critical program sections are easily recognized thanks to color coding. The analysis results thus provide valuable feedback in optimizing the stack usage of an application. The predicted worst-case stack usages of individual tasks in a system can be used in an automated overall stack usage analysis for all tasks running on one Electronic Control Unit, as described in [7] for systems managed by an OSEK/VDX real-time operating system.

4 WCET Analysis: Worst-Case Execution Time Prediction

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Yet the determination of the Worst-Case Execution Time (WCET) of a task is a difficult problem because of the characteristics of modern software and hardware [17].

Embedded control software (e.g., in the automotive industries) tends to be large and complex. The software in a single electronic control unit typically has to provide different kinds of functionality. It is usually developed by several people, several groups or even several different providers. Code generator tools are widely used. They usually hide implementation details to the developers and make an understanding of the timing behavior of the code more difficult. The code is typically combined with third party software such as real-time operating systems and/or communication libraries.

Concerning hardware, there is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance micro-controllers and DSPs). Pipelines enable acceleration by overlapping the executions of different instructions. Consequently the execution behavior of the instructions cannot be analyzed separately since it depends on the execution history.

Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) real-time systems, this is undesirable and possibly even hazardous. Making the safe yet—for the most part—unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

The widely used classical methods of predicting execution times are not generally applicable. Software monitoring or the dual-loop benchmark change the code, which in turn has impact on the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for one input. They cannot be used to infer the execution times for all possible inputs in general.

Furthermore, the execution time depends on the processor state in which the execution is started. Modern processor architectures often violate implicit assumptions on the worst start state. The reason is that they exhibit timing anomalies as defined in [9], which consist of a locally advantageous situation, e.g., a cache hit, resulting in a globally larger execution time. As also demonstrated in [9], processor pipelines may exhibit so-called domino effects where—for some special pieces of code—the difference between two start states of the pipeline does not disappear over time, but leads to a difference in execution time that cannot be bounded by a constant.

4.1 Structure of WCET Computation

Abstract interpretation can be used to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a program point. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path. This approach can help to overcome the challenges listed above.

AbsInt's WCET tool **aiT** determines the WCET of a program task in several phases [5] (see Figure 3):

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from a binary program;
- **Value Analysis** computes value ranges for registers and memory cells, and address ranges for instructions accessing memory;
- **Loop Bound Analysis** determines upper bounds for the number of iterations of simple loops;
- **Cache Analysis** classifies memory references as cache misses or hits [4];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [8];
- **Path Analysis** determines a worst-case execution path of the program [16].

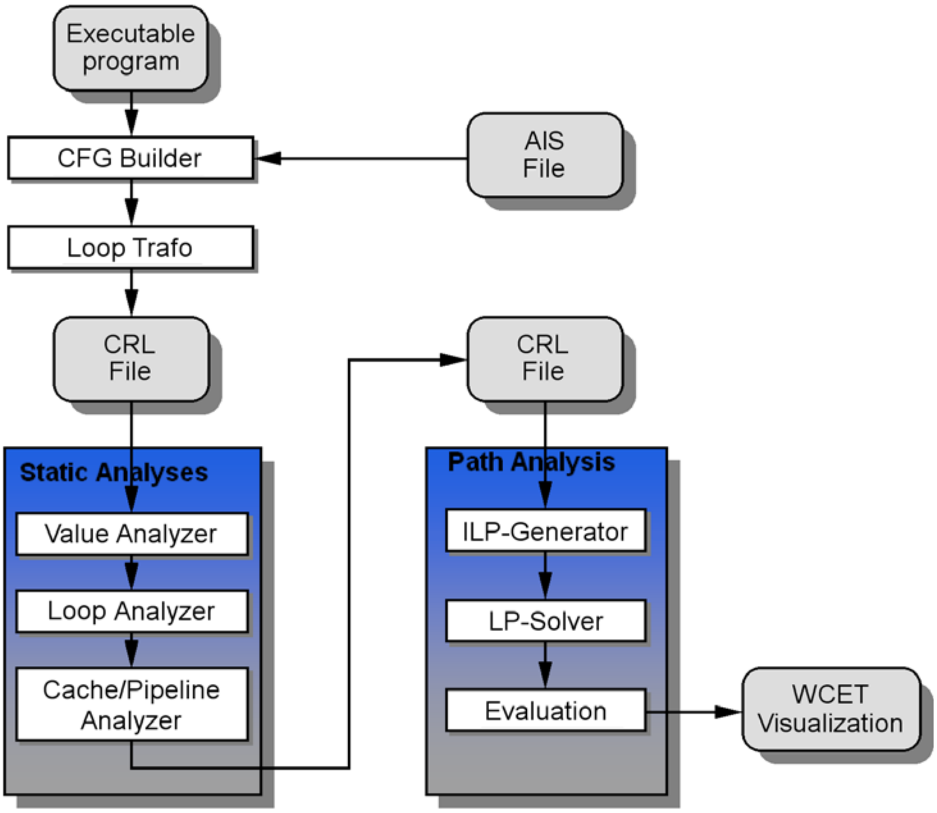


Figure 3: Phases of WCET computation

Separating WCET determination into several phases makes it possible to use different methods tailored to the subtasks [16]. Value analysis, cache analysis, and pipeline analysis are based on abstract interpretation [2]. Integer linear programming is used for path analysis.

aiT allows to inspect the timing behavior of (time-critical parts of) program tasks. The analysis results are determined without the need to change the code and hold for all executions (for the intrinsic cache and pipeline behavior). **aiT** takes into account the combination of all the different hardware characteristics while still obtaining tight upper bounds for the WCET of a given program in reasonable time.

4.2 Reconstruction of the Control Flow from Binary Programs

The starting point of our analysis framework (see Figure 3) is a binary program and a so-called *AIS file* containing additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc. In the first step a decoder reads the executable and reconstructs the control flow [13, 14]. This requires some knowledge about the underlying hardware, e.g., which instructions represent branches or calls. The reconstructed control flow is annotated with the information needed by subsequent analyses and then translated into CRL (Control-Flow Representation Language)—a human-readable intermediate format designed to simplify analysis and optimization at the executable/assembly level. This annotated control-flow graph serves as the input for micro-architecture analysis.

The decoder can find the target addresses of absolute and pc-relative calls and branches, but may have difficulties with target addresses computed from register contents. Thus, **aiT** uses specialized decoders that are adapted to certain code generators and/or compilers. They usually can recognize branches to a previously stored return address, and know the typical compiler-generated patterns of branches via switch tables. Yet non-trivial applications may still contain some computed calls and branches (in hand-written assembly code) that cannot be resolved by the decoder; these unresolved computed calls and branches are documented by appropriate messages and require user annotations. Such annotations may list the possible targets of computed calls and branches, or tell the decoder about the address and format of an array of function pointers or a switch table used in the computed call or branch.

4.3 Value Analysis

Value analysis as described in Section 2 tries to determine the values in the processor memory for every program point and execution context. Its results are used to determine possible addresses of indirect memory accesses—important for cache analysis—and in loop bound analysis. They are usually so good that only a few indirect accesses cannot be determined exactly. Address ranges for these accesses may be provided by user annotations.

4.4 Loop Bound Analysis

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. **aiT** tries to determine the number of loop iterations by *loop bound analysis*, but succeeds in doing so for simple loops only. Bounds for the iteration numbers of the remaining loops must be provided as user annotations.

aiT employs two different methods for loop bound analysis. The older method relies on a combination of value analysis and pattern matching, which looks for typical loop patterns. In general, these loop patterns depend on the code generator and/or compiler used and sometimes even on the optimization level.

The newer method described in [3] uses an interprocedural data-flow analysis to derive loop invariants from the semantics of the instructions. This new analysis does not depend on the compiler used or optimization level, but only on the semantics of the instruction set for the target machine. It is able to handle loops with multiple exits and multiple modifications of the loop counter per iteration including modifications in procedures called from the loop.

4.5 Cache Analysis

Cache analysis classifies the accesses to main memory. The analysis in our tool is based upon [4], which handles analysis of caches with LRU (Least Recently Used) replacement strategy. However, it had to be modified to reflect the non-LRU replacement strategies of common microprocessors: the pseudo-round-robin replacement policy of the ColdFire MCF 5307, and the PLRU (Pseudo-LRU) strategy of the PowerPC MPC 750 and 755. The modified algorithms distinguish between sure cache hits and unclassified accesses. The deviation from perfect LRU is the reason for the reduced predictability of the cache contents in case of ColdFire 5307 and PowerPC 750/755 compared to processors with perfect LRU caches [6].

4.6 Pipeline Analysis

Pipeline analysis models the pipeline behavior to determine execution times for sequential flows (basic blocks) of instructions, as done in [11]. It takes into account the current pipeline state(s), in particular resource occupancies, contents of prefetch queues, grouping of instructions, and classification of memory references by cache analysis. The result is an execution time for each basic block in each distinguished execution context.

Like value and cache analysis, pipeline analysis is based on the framework of abstract interpretation. Pipeline analysis of a basic block starts with a set of pipeline states determined by the predecessors of the block and lets this set evolve from instruction to instruction by a kind of cycle-wise simulation of machine instructions. In contrast to a real simulation, the abstract execution on the instruction level is in general non-deterministic since information determining the evolution of the execution state is missing, e.g., due to non-predictable cache contents. Therefore, the abstract execution of an instruction may cause a state to split into several successor states. All the states computed in such tree-like structures form the set of entry states for the successor instruction. At the end of the basic block, the final set of states is propagated to the successor blocks. The described evolution of state sets is repeated for all basic blocks until it stabilizes, i.e. the state sets do not change any more.

The output of pipeline analysis is the number of cycles a basic block takes to execute, for each context, obtained by taking the upper bound of the number of simulation cycles for the sequence of instructions for this basic block. These results are then fed into path analysis to obtain the WCET for the entire task.

4.7 Path Analysis

Using the results of the micro-architecture analyses, path analysis determines a safe estimate of the WCET. The program's control flow is modeled by an integer linear program [16, 15] so that the solution to the objective function is the predicted worst-case execution time for the input program. A special mapping of variable names to basic blocks in the integer linear program enables execution and traversal counts for every basic block and edge to be computed.

4.8 Analysis of Loops and Recursive Procedures

Loops and recursive procedures are of special interest since programs spend most of their runtime there. Treating them naively when analyzing programs for their cache and pipeline behavior results in a high loss of precision.

Frequently the first execution of the loop body loads the cache, and subsequent executions find most of their referenced memory blocks in the cache. Because of speculative prefetching, cache contents may still change considerably during the second iteration. Therefore, the first few iterations of the loop often encounter cache contents quite different from those of later iterations. Hence it is useful to distinguish the first few iterations of loops from the others. This is done in the VIVU approach (virtual inlining, virtual unrolling) [10].

Using upper bounds on the number of loop iterations, the analyses can virtually unroll not only the first few iterations, but all iterations. The analyses can then distinguish more contexts and the precision of the results is increased—at the expense of higher analysis times.

4.9 Usage of aiT

The techniques described above have been incorporated into **AbsInt**'s **aiT** WCET analyzer tools. They get as input an executable, user annotations, a description of the (external) memories and buses (i.e. a list of memory areas with minimal and maximal access times), and a task (identified by a start address). A task denotes a sequentially executed piece of code (no threads, no parallelism, and no waiting for external events). This should not be confused with a task in an operating system that might include code for synchronization or communication.

The WCET analyzers compute an upper bound of the runtime of the task (assuming no interference from the outside). Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately (e.g., by a quantitative analysis).

The task WCETs predicted by **aiT** can be used to determine an appropriate scheduling scheme for the tasks and to perform an overall schedulability analysis in order to guarantee that the application meets all timing constraints (also called *timing validation*) [12]. Some real-time operating systems offer tools for schedulability analysis, but all these tools require the WCETs of tasks as input.

4.10 Visualization of aiT's Results

aiT's results are written into a report file. In addition, **aiT** produces a picture description that can be visualized by the **aiSee** tool [1] to view detailed information delivered by the analysis.

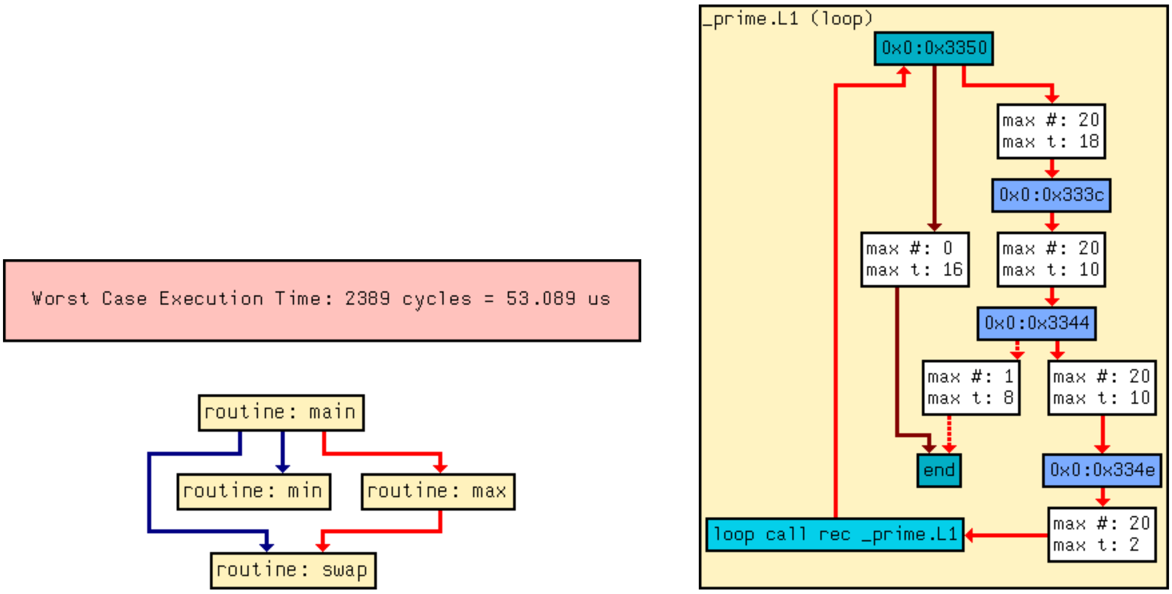


Figure 4: Call graph and control-flow graph with WCET results

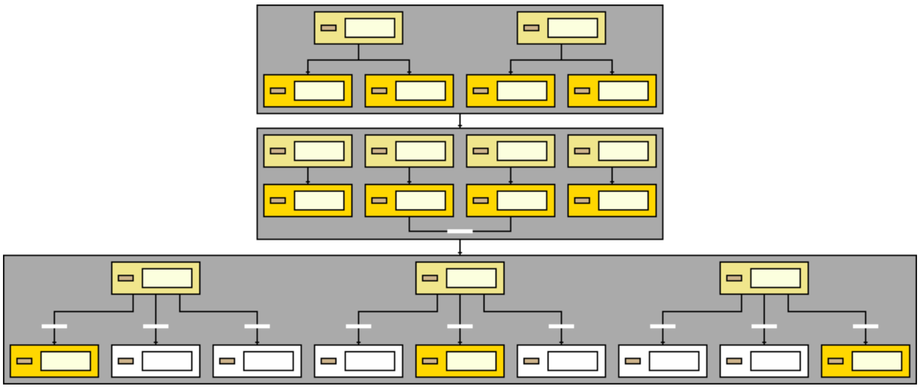


Figure 5: Possible pipeline states in a basic block

Figure 4, left, shows the graphical representation of the call graph for some small example. The calls (edges) that contribute to the worst-case runtime are marked by the color red. The computed WCET is given in CPU cycles and in microseconds provided that the cycle time of the processor has been specified.

Figure 4, right, shows the basic block graph of a loop. The number `max #` describes the maximal number of traversals of an edge in the worst case, while `max t` describes the maximal execution time of the basic block from which the edge originates (taking into account that the basic block is left via the edge). The worst-case path, the iteration numbers and timings are determined automatically by **aiT**.

Figure 5 shows the possible pipeline states for a basic block in this example. Such pictures are shown by **aiT** upon special demand. The large dark grey boxes correspond to the instructions of the basic block, and the smaller rectangles in them stand for individual pipeline states. Their cyclewise evolution is indicated by the strokes connecting them. Each layer in the trees corresponds to one CPU cycle. Branches in the trees are caused by conditions that could not be statically evaluated, e.g., a memory access with unknown address in presence of memory areas with different access times. On the other hand, two pipeline states fall together when details they differ in leave the pipeline. This happened for instance at the end of the second instruction, reducing the number of states from four to three.

Figure 6 shows the top left pipeline state from Fig. 5 in greater magnification. It displays a diagram of the architecture of the CPU (in this case a PowerPC 555) showing the occupancy of the various pipeline stages with the instructions currently being executed.

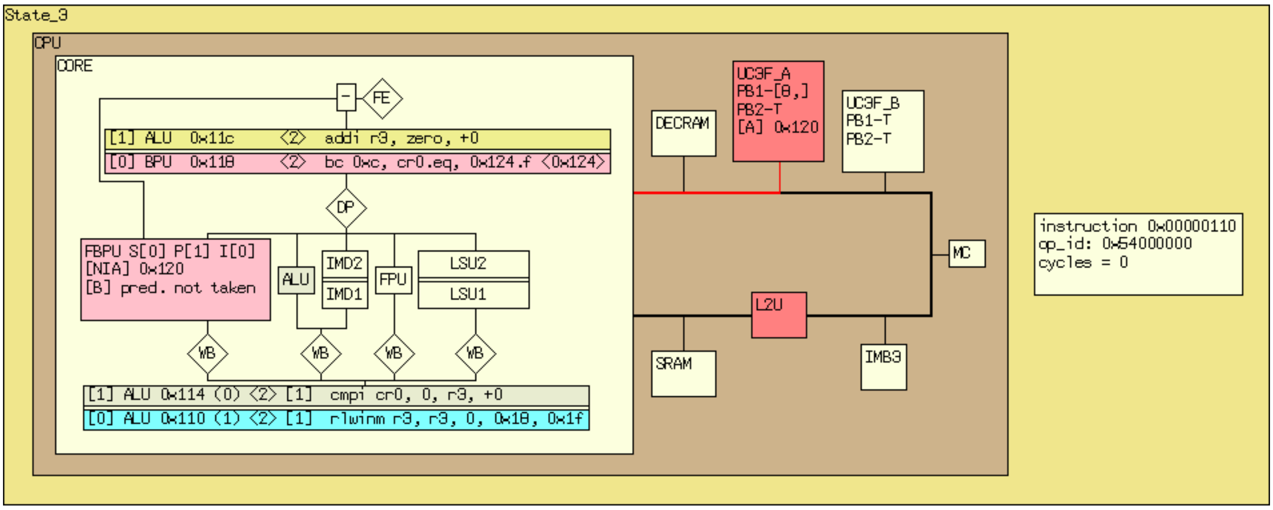


Figure 6: Individual pipeline state

5 Dependence on Target Architectures

There are **aiT** versions for PowerPC MPC 555, 565, and 755, ColdFire 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, and Tricore 1.3.

Decoders are automatically generated from processor specifications defining instruction formats and operand meaning. The CRL format used for describing control-flow graphs is machine-independent. *Value Analysis* must interpret the operations of the target processor. Hence, there is a separate value analyzer for each target, but features shared by many processors (e.g., branches based on condition bits) allowed for considerable code sharing among the various value analyzers.

There is only one cache analyzer with a fixed interface to pipeline analysis. It is parameterized on cache size, line size, associativity, and replacement strategy. Each replacement strategy supported by **aiT** is implemented by a table for line age updates that is interpreted by the cache analyzer.

The pipeline analyzers are the most diverse part of **aiT**. The supported target architectures are grouped according to the complexity of the processor pipeline. For each group a common conceptual and coding framework for pipeline analysis has been established, in which the actual target-dependent analysis must be filled in by manual coding.

6 Precision of aiT

Since the real WCET is not known for typical real-life applications, statements about the precision of **aiT** are hard to obtain. For an automotive application running on MPC 555, one of **AbsInt**'s customers has observed an overestimation of 5–10% when comparing **aiT**'s results and the highest execution times observed in a series of measurements (which may have missed the real WCET). For an avionics application running on MPC 755, Airbus has noted that **aiT**'s WCET for a task typically is about 25% higher than some measured execution times for the same task, the real but non-calculable WCET being in between. Measurements at **AbsInt** have indicated overestimations ranging from 0% (cycle-exact prediction) till 10% for a set of small programs running on M32C, TMS320C33, and C16x/ST10. Table 1 shows the results for C166. The analysis times were moderate—a few seconds till about 3 minutes for edn.

Table 1: Precision of **aiT** for some C166 programs

Example		from external RAM			from flash		
Program	Size	measured cycles	predicted cycles	over-estimation	measured cycles	predicted cycles	over-estimation
fac	2.9k	949	960	1.16 %	810	832	2.72 %
fibo	3.4k	2368	2498	5.49 %	2142	2228	4.01 %
coverc1	16k	5670	5672	0.04 %	3866	4104	6.16 %
coverc	4.3k	7279	7281	0.03 %	5820	6202	6.56 %
morswi	5.9k	17327	17332	0.03 %	8338	8350	0.14 %
coverc2	24k	18031	18178	0.82 %	12948	14054	8.54 %
swi	24k	18142	18272	0.72 %	13330	14640	9.83 %
edn	13k	262999	267643	1.77 %	239662	241864	0.92 %

7 Conclusion

Tools based on abstract interpretation can perform static program analysis of embedded applications. Their results hold for all program runs with arbitrary inputs. Employing static analyzers is thus orthogonal to classical testing, which yields very precise results, but only for selected program runs with specific inputs.

aiT allows to inspect the timing behavior of (time-critical parts of) program tasks. It takes into account the combination of all the different hardware characteristics while still obtaining tight upper bounds for the WCET of a given program in reasonable time. **StackAnalyzer** and **aiT** are used among others by Airbus in the development of various safety-critical applications for the A380. They will be used as verification tools in the sense of DO178b. The qualification requirements for such verification tools are lighter than for code generation tools. In contrast to code generation tools, verification tools cannot introduce any errors into the safety-critical system. Failure of a verification tool may only lead to an overlooked error.

In a less regulated application area, **StackAnalyzer** and **aiT** can be used for optimizing the system. For instance, the results of **StackAnalyzer** are useful when optimizing the assignment of priorities to tasks in order to minimize the memory consumption in an OSEK-like operating system. The results of **aiT** can be used to find an optimal schedule in a time-triggered operating system.

The usage of static analyzers enables one to develop complex systems on state-of-the-art hardware, increases safety, and saves development time. Precise stack usage and timing predictions enable the most cost-efficient hardware to be chosen.

References

[1] AbsInt Angewandte Informatik GmbH. *aiSee Home Page*. http://www.aisee.com, 2006.

[2] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, California, 1977.

[3] Christoph Cullmann. Statische Berechnung sicherer Schleifengrenzen auf Maschinencode. Master's thesis, Universität des Saarlandes, 2006.

[4] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[5] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[6] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[7] Winfried Janz. Das OSEK Echtzeitbetriebssystem, Stackverwaltung und statische Stackbedarfsanalyse. In *Embedded World*, Nuremberg, Germany, February 2003.

[8] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[9] Thomas Lundquist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, December 1999.

[10] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, March /April 1998.

[11] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34, pages 35–44, May 1999.

[12] John A. Stankovic. *Real-Time and Embedded Systems*. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. http://www-ccs.cs.umass.edu/sdcr/rt.ps.

[13] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.

[14] Henrik Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 112–120, Snowbird, Utah, USA, June 2001.

[15] Henrik Theiling. ILP-based interprocedural path analysis. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Proceedings of EMSOFT 2002, Second International Conference on Embedded Software*, volume 2491 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2002.

[16] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.

[17] Reinhard Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1 – 14–23. CRC Press, 2005.

Experiences in Quality Checking Medical Guidelines using Formal Methods

Perry Groot and **Arjen Hommersom** and **Peter Lucas**[1]
Michael Balser and **Jonathan Schmitt**[2]

Abstract. In health care, the trend of evidence-based medicine, has led medical specialists to develop medical guidelines, which are large nontrivial documents suggesting the detailed steps that should be taken by health-care professionals in managing the disease in a patient. In the Protocure project the objective has been to assess the improvement of medical guidelines using formal methods. This paper reports on some of our findings and experiences in quality checking medical guidelines. In particular the formalisation of meta-level quality criteria for good practice medicine, which is used in conjunction with medical background knowledge to verify the quality of a guideline dealing with the management of diabetes mellitus type 2 using the interactive theorem prover KIV. For comparison, analogous investigations have been performed with other techniques including automatic theorem proving and model checking.

1 Introduction

Computer-based decision support in health-care is a field with a long standing tradition, dealing with complex problems in medicine such as diagnosing disease and assisting in the prescription of appropriate treatment. The trend of the last decades has been to base clinical decision making more and more on sound scientific evidence, i.e; this has been called *evidence-based medicine* [41, 45]. In practice this has led organisations of medical specialists in particular areas to develop medical guidelines, i.e., structured documents suggesting the detailed steps that should be taken by health-care professionals in managing the disease of a patient, to promote standards of medical care. Ethical concerns about evidence-based medicine have been raised [11] and there is a potential risk that medical guidelines do harm when improperly developed [44]. However, guidelines have also shown to improve health-care outcomes [44] and may even reduce the costs of care up to 25% [8].

Researchers in Artificial Intelligence have picked up on the increasing use of medical guidelines and are working towards offering computer-based support in the development and deployment of guidelines using computer-oriented languages and tools [10, 30]. This has given rise to the emergence of a new paradigm for the modelling of complex clinical processes as a ‘network of tasks’, where a task consists of a number of steps, each step having a specific function or goal [15, 28]. Examples of languages that support task models, and which have been evolving since the 1990s, include PRO*forma* [16, 17], Asbru [37, 40], EON [42, 43], and GLIF3 [28]. In this work, medical guidelines are considered as real-world examples of structured documents, which can benefit from formalisation, although experience has shown that looking upon medical guidelines as formal objects is a nontrivial task [29].

One of the reasons for this is that medical guidelines should not be considered static objects as they are changed on a regular basis as new scientific evidence becomes available. Rapidly changing and evolving evidence makes it difficult to adjust guidelines in such a way as to keep them up to date. As a consequence, computer-based support of guideline development should also be concerned with the updating of guidelines, i.e., indicate where guidelines should be updated in light of new evidence.

In this article, we approach this problem by applying formal methods to checking the quality of medical guidelines. Here, we are mainly concerned with checking of *general* quality criteria of good practice medicine a guideline should comply to. This has been called the meta-level approach to quality checking of medical guidelines [24]. For example, a guideline should preclude the prescription of redundant drugs, or advise against a prescription of a treatment that is less effective than some alternative. Newly obtained evidence may invalidate properties of a guideline, because, for example, new patient management options have arisen or financial costs have decreased through new developments in drug therapy.

A solid foundation for the application of formal methods to the quality checking of medical guidelines can already be found in literature. In [15, 25] logical methods have been used to analyse properties of guidelines. We have shown in [24] that the theory of abductive diagnosis can be taken as a foundation for the formalisation of quality requirements of a medical guideline in temporal logic. This result has been used in verifying quality requirements of good practice medicine of alternative treatments [21].

The contribution of this paper, is that we formalise quality requirements of medical guidelines which include, besides separate treatments, also the temporal relations between separate treatments, by which we mean the order in which they are prescribed. Second, using our quality requirements and medical background knowledge, we interactively verify a guideline dealing with the management of diabetes mellitus type 2. More specifically, we model the guideline as a ‘network of tasks’ using the language Asbru and, additionally, verify meta-level properties for this model using KIV, an interactive theorem prover [6]. To the best of our knowledge, verification of a fully formalised guideline, as a network of tasks, using medical background knowledge has not been done before. The presented framework provides a sound formal foundation for further research in quality checking of medical guidelines and the temporal relations among different treatments involved.

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen, e-mail:{perry,arjenh,peterl}@cs.ru.nl

[2] Institut für Informatik, Universität Augsburg, D-86135 Augsburg, e-mail: {balser,jonathan.schmitt}@informatik.uni-augsburg.de

The remainder of this paper is structured as follows. Section 2 gives an introduction to the Protocure project and the methodology employed within the project.[3] Section 3 gives an introduction to medical guidelines. Section 4 gives an overview of Asbru, the guideline representation language used throughout our work. Section 5 discusses in more detail the approach to formal verification of medical guideline. It discusses the main elements of a guideline a formal language should address and discusses the three types of knowledge involved: background knowledge, the treatment order in the guideline, and the quality requirements. Section 6 discusses in more detail how to formalise these three knowledge types in the context of diabetes mellitus type 2. Section 7 discusses in more detail how to translate everything into the KIV system. Section 8 gives the results with interactive verification with the theorem prover KIV.

2 Protocure: Improving medical guidelines by formal methods

The aim of the Protocure project has been to take the formalisation of guidelines one step further, by using guideline representation languages for modelling medical guidelines as formal objects and integrating them with formal methods for quality checking. The main objective of the Protocure project was the assessment of guideline improvement using formal methods, which has been done using the methodology shown in Figure 1 [2]. Initially, a medical guideline is selected, which is then gradually transformed into a formal representation. This transformation basically consists of two phases. Firstly, the guideline is modelled in the Asbru language, which is a language specifically designed for the modelling of medical guidelines. Asbru is described in detail in Section 4. Secondly, the Asbru model of the guideline is transformed in a formal language that can be used for verification. Formal languages, tools, and techniques that have been used within the Protocure project are (1) KIV, an interactive theorem prover that uses a variant of temporal logic, (2) Otter, an automatic theorem prover, and (3) SMV, a model checker that uses computation tree logic and linear temporal logic. These are described in more detail in forthcoming sections.

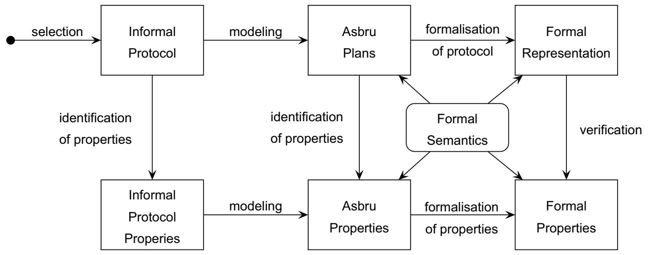

selection
Informal Protocol
modeling
Asbru Plans
formalisation of protocol
Formal Representation
identification of properties
identification of properties
Formal Semantics
verification
Informal Protocol Properies
modeling
Asbru Properties
formalisation of properties
Formal Properties

Figure 1. The process of guideline formalisation and verification as done in the Protocure project.

Closely related to the modelling of the guideline is the modelling of the properties one wants to check for the guideline under study. Several sources can be used to obtain such properties, which then also need to be translated into a formal language that will be used for verification. The simplest properties, so-called *structural properties* [12], are those properties that ensure that the Asbru model created is correct, e.g., reachability of all states. More complex properties deal with the medical intentions one wants to obtain when using a guideline. These can be derived from the guideline text or for example from quality indicators independently developed from the guideline [18]. Such properties need interpretation and were found to be harder to formalise. In this paper, we look, among others, at a specific type of such complex properties, namely meta-level quality requirements, which state requirements for general good medical practice.

3 Medical guidelines

Guidelines, medical guidelines, or practice guidelines are all commonly used abbreviations for the full term ‘clinical practice guideline’. An often cited definition of guidelines is the one by Field and Lohr [14]:

> Clinical practice guidelines are systematically developed statements to assist practitioner and patient decisions about appropriate health care for specific clinical circumstances.

Though ‘protocol’ is often synonymously used for ‘guideline’, a protocol gives detailed statements about *how* one should act in daily practice, whereas a guideline gives more general scientifically founded statements about *what* should be done. Protocols are often seen as more detailed, practice-oriented versions of a guideline [27]. In this work the focus is on medical guidelines.

An example of a fragment of a guideline is shown in Figure 2. It is part of the guideline for general practitioners about the treatment of diabetes mellitus type 2 [34]. General practitioners’ guidelines are normally quite compact. Guidelines for medical specialists are often large – they can be as large as 100 pages – but even then they consists of sections similar to our example. Translating a guideline into a clear and structured fragment such as in Figure 2 can take a lot of effort; however, the formalisation of a guideline is not the main focus of the work presented, which is about verification of a formalised guideline.

- Step 1: diet.
- Step 2: if Quetelet index (QI) ≤ 27, prescribe a sulfonylurea drug; otherwise, prescribe a biguanide drug.
- Step 3: combine a sulfonylurea drug and biguanide (replace one of these by a α-glucosidase inhibitor if side-effects occur).
- Step 4: one of the following:
  - ● oral antidiabetic and insulin
  - ● only insulin

Figure 2. Tiny fragment of a clinical guideline on the management of diabetes mellitus type 2. If one of the steps $k = 1, 2, 3$ is ineffective, the management moves to step $k + 1$

The diabetes mellitus type 2 guideline provides practitioners with a clear structure of recommended actions to be taken for the control of the glucose level. This kind of information is typically found in medical guidelines in the sense that medical knowledge is combined with information about order and time of treatment (e.g., sulfonylurea in step 2), about patients and their environment (e.g., Quetelet index lower than or equal to 27), and finally which drugs are to be administered to the patient (e.g., a sulfonylurea drug). When verifying the quality of a guideline, the formal language used should at least address these elements. We come back to these elements in more detail in Section 5.1.

4 Medical guidelines in Asbru

Much research has already been devoted to the development of representation languages for medical guidelines. Most of them look at

[3] http://www.protocure.org

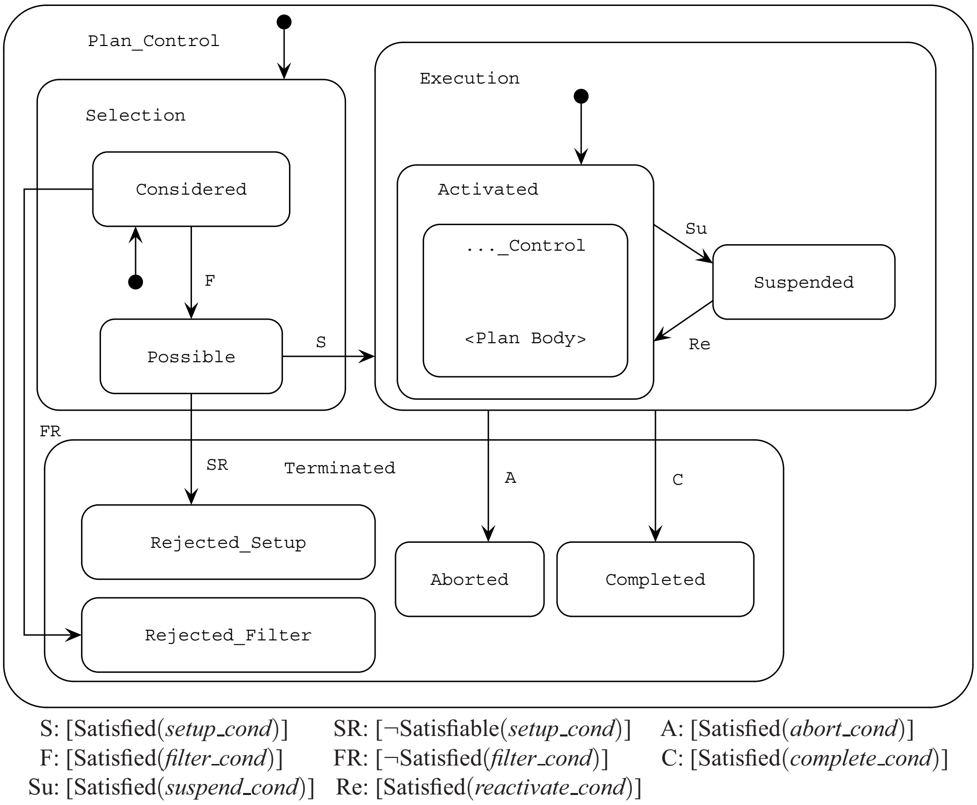

Plan_Control
Selection
Considered
F
Possible
S
Execution
Activated
..._Control
<Plan Body>
Su
Suspended
Re
FR
SR
Terminated
A
C
Rejected_Setup
Rejected_Filter
Aborted
Completed
S: [Satisfied(*setup_cond*)]
SR: [¬Satisfiable(*setup_cond*)]
A: [Satisfied(*abort_cond*)]
F: [Satisfied(*filter_cond*)]
FR: [¬Satisfied(*filter_cond*)]
C: [Satisfied(*complete_cond*)]
Su: [Satisfied(*suspend_cond*)]
Re: [Satisfied(*reactivate_cond*)]

Figure 3. The plan state model, where Satisfied(*cond*) denotes that the environment satisfied the condition *cond* whereas Satisfiable(*cond*) denotes that, theoretically, the environment could still satisfy the condition *cond*, i.e., that no deadline has passed in case of time constraints.

guidelines consisting of a composition of actions, whose execution is controlled by conditions [27]. However, most of them are not formal enough for the purpose of our research as they often incorporate free-text elements which do not have a clear semantics. Exceptions to this are PRO*forma* [16, 17] and Asbru [37, 40]. The latter has been chosen in our research as a basis to formalise a medical guideline.

4.1 Introduction to Asbru

A medical guideline is considered in Asbru as a hierarchical *plan*. The main components of an Asbru plan are intentions, conditions, plan-body, and time annotations. Furthermore, a plan can have arguments and can alter the value of variables.

The *intentions* are the high-level goals of a plan. Intentions can be expressed in terms of achieving, maintaining, or avoiding a certain state or action. The states or actions to which intentions refer can be intermediate or final (overall). In total there are twelve possible forms of intentions built up by combining elements from the sets {achieve, maintain, avoid}, {intermediate, overall}, and {state, action}.

Conditions can be associated to a plan to define different aspects of its execution. The most important types of condition are: (1) filter and setup conditions,[4] which must be true before a plan can start, (2) abort conditions, which define when a plan must abort, and (3) complete conditions, which define when a started plan finishes successfully. Conditions can be 'over-ridable' (i.e., health personnel can manually satisfy the condition) or 'require confirmation' (i.e., conditions must be explicitly confirmed before they are satisfied).

The *plan-body* contains the actions, sub-plans, or both to be executed as part of the plan. The main types of plan-body are: (1) user-performed: an action has to be performed by a user, which requires interaction, which is not further modelled, (2) single-step: an action which can be either an activation of a sub-plan, an assignment of a variable, a request for an input value, or an if-then-else statement, (3) sub-plans: a set of plans to be performed in a given order, either sequentially, in parallel, in any-order, or unordered, and (4) cyclical plans: a repetition of actions over a time period. In case of sub-plans, it is also required to specify a waiting strategy to describe which of the sub-plans must be completed for the super plan to complete, e.g., all sub-plans should be executed (**wait-for all**).

Time annotations can be associated to various Asbru elements, e.g., intentions, conditions, plan activations. A time annotation specifies (1) in which interval things must start, (2) in which interval things must end, (3) their minimal and maximal duration, and (4) a reference time point.

4.2 The semantics of Asbru

To help in the understanding of Asbru we review here the semantics of Asbru in a semi-formal statechart notation [5]. In Asbru, plans are organised in a hierarchy, where a plan may include a number of sub-plans. The semantics of Asbru is defined in [3] by flattening the hierarchy of plans and using one top level control to execute all plans synchronously. Within each top level step, a step of every plan is executed. Whether a plan is able to progress depends on its conditions.

[4] filter conditions are conditions about values that cannot change value, e.g., $sex = male$, whereas setup conditions are conditions about values that may change, e.g., glucose level.

The plan state model shown in Figure 3 defines the semantics of the main plan hierarchy. The 'Plan_Control' is divided into a selection phase, an execution phase, and a termination phase. Each plan goes into the 'Considered' state when it receives a *consider* signal. In this state its *filter condition* is checked. If it evaluates to true, control advances to the state 'Possible'. Then the setup condition is checked and if it is passed, control advances to the execution phase. If the filter condition is not satisfied or the setup condition is not satisfiable anymore (i.e., it is not possible to satisfy the condition in the future, because a deadline has passed), the plan is rejected. The same happens, if the super-plan terminates. In the execution phase the plan waits for an external signal *activate*, to be sent by its super-plan.

In state 'Activated', the sub-plans are executed, which can be sequentially, in parallel, unordered, or in any order, and each order determines a different controlling statechart [3]. A plan can synchronise its sub-plans using the signals *consider* and *activate*. Additional control to propagate execution states of a sub-plan to its parent and vice versa is also present, e.g., the abortion of a mandatory sub-plan enforces the parent-plan also to abort. Sub-plans can either be completed successfully or aborted, e.g., in the case of emergency patient readings.

The complete technical definitions, in addition to the semantics of the other constructs that are not shown here, can be found in [5].

5 Verification of medical guidelines

5.1 Requirements for the verification of guidelines

To be able to verify quality criteria of medical guidelines using formal methods, we need to have a language that can be used to express quality criteria that can be related to the key elements in a guideline. In Section 3, we stated that the key elements in medical guidelines are (at least) order in time, patients, and interventions. Here, we discuss our choices for a language for the formal representation of those key elements, used in the remainder of the paper.

Time: As medical management is a time-oriented process, diagnostic and treatment actions described in guidelines are performed in a temporal setting. It has been shown previously that the step-wise, possibly iterative, execution of a guideline can be described by means of temporal logic [25]. This is a modal logic [13], where relationships between worlds in the usual possible-world semantics of modal logic is understood as time order. In this paper, we will use a variant of this logic, based on future-time linear temporal logic. The language of this logic is first-order logic augmented with the temporal operators listed in Table 1. The semantics of this language is given by a set D, representing the universe of discourse, a set of interpretations I_t for interpreting statements from the first-order logic, and a function *succ*, where $succ(t)$ is the set of zero or one successors of time points of t. First-order expressions φ at time t are interpreted using I_t in the domain D; for example, $t \vDash \varphi$ means that φ is satisfied at time t w.r.t. I_t and D [13].

Note that the **last** modality can only hold in models where at some point following the successor function, no successor exists. In all other models, **last** will never hold. Also note that some operators can be defined in terms of other operators, e.g., $\Box\,\varphi \equiv \neg \Diamond \neg \varphi$ and **last** $\equiv \bullet\, \bot$. A more expressive logic can be gained by including, for example, the **until** operator, where φ **until** ψ denotes that eventually ψ holds and before that φ holds. However, as such operators are not used in this paper, they have been omitted.

This logic allows one to look at guidelines formally at a particular abstraction level. In Section 8, we show this logic to be suitable for quality checking of medical guidelines; however, it is possible to add more fine-grained temporal operators if they are needed.

Patient groups: Although in practice a guideline is used for the management of a particular patient, recommendations in guidelines are always written with a certain *patient group* in mind – not just a single patient. Patient groups are groups of patients that share common characteristics about their current state or previous states. One can abstract from the actual situation of a patient by providing a logical language that refers to one or more situations, including the necessary common characteristics, without fixing all the details. Typical elements for describing the state of patients are symptoms, signs, and test outcomes. Here we have chosen to use predicate logic with equality and unique names assumption [32]. For example, the literal 'Condition$(\textit{hyperglycaemia})$' is used to represent the patient group of all patients that currently have the condition of hyperglycaemia. Subgroups of patient groups can be specified by using a conjunction with additional literals, e.g., 'Condition$(\textit{hyperglycaemia}) \wedge \text{QI} \leq 27$' specifies the patient group of patients who have hyperglycaemia and also have a Quetelet index less than or equal to 27. We sometimes represent the conjunction also in set form, e.g., the latter conjunction becomes '$\{\text{Condition}(\textit{hyperglycaemia}), \text{QI} \leq 27\}$'.

Interventions and treatments: An intervention is the act of intervening, interfering, or interceding with the intent of modifying the outcome. In medicine, interventions include all medical actions that influence the state of a patient or his environment. A treatment is usually restricted to methods that provide a cure for an illness or disability, however, the terms intervention and treatment are often used synonymously. We have chosen to represent the domain of interventions by a countable set. Subsets of this set are interpreted as *treatments* in which each intervention of the set is applied. Interventions which are not an element of the treatment are assumed not to be applied. We abstract from medical management details such as changing drug dosages.

5.2 Verification approach

Medical guidelines give recommendations based on the best available evidence. Although diabetes mellitus type 2 is a complicated disease, the guideline fragment shown in Figure 2 is not. This indicates that much knowledge concerning diabetes mellitus type 2 is missing from the guideline. Verifying whether a guideline fulfils some property therefore additionally needs the specification of *background knowledge*.

The ideas that we use here to verify quality requirements for medical guidelines are inspired by previous work, where a distinction was made between the different types of knowledge that are involved in defining quality requirements [21]. We assume that there are at least three types of knowledge involved in detecting the violation of good medical practice:

1. Knowledge concerning the (patho)physiological mechanisms underlying the disease, and the way treatment influences these mechanisms. The knowledge involved could be for example causal or empirical in nature, and is an example of *object-knowledge*.
2. Knowledge concerning the recommended treatment in every step of the guideline and how the choice for each treatment is affected by the state of the patient, i.e., the order information from the medical guideline. This is also an example of *object-knowledge*.

Table 1. Used temporal operators; t stands for a time instance

Notation	Interpretation	Formal semantics
$\Box\,\varphi$	φ will always be true	$t \vDash \Box\,\varphi \Leftrightarrow \forall t' \geq t : t' \vDash \varphi$
$\Diamond\,\varphi$	φ will eventually be true	$t \vDash \Diamond\,\varphi \Leftrightarrow \exists t' \geq t : t' \vDash \varphi$
$\circ\,\varphi$	execution does not terminate and the next state satisfies φ	$t \vDash \circ\,\varphi \Leftrightarrow \exists\, t' \in \mathit{succ}(t) : t' \vDash \varphi$
$\bullet\,\varphi$	either execution terminates or the next state satisfies φ	$t \vDash \bullet\,\varphi \Leftrightarrow \forall\, t' \in \mathit{succ}(t) : t' \vDash \varphi$
last	the current state is the last	$t \vDash$ **last** $\Leftrightarrow \mathit{succ}(t) = \varnothing$

3. Knowledge concerning good practice in treatment selection; this is *meta-knowledge*.

The first type of object-knowledge will be called *background knowledge*. The second type of object-knowledge is the order information from the medical guideline, which can be considered a network of tasks or a hierarchical plan. The plan prescribes treatment which influences the (patho)physiological mechanisms, which results in information about patient groups that can be used by the plan to make the best possible decision in subsequent step of the protocol. Incompleteness of background knowledge may lead to insufficient knowledge about a patient, which may result in a plan making a non-deterministic choice. Of course, the guideline should recommend the collection of data when possible if this data is crucial for decision making.

The third type of knowledge, the meta-knowledge, includes general knowledge about good practice medicine, for example, preferring a treatment over another if it uses a smaller number of drugs and has an equal effect on the patient. This knowledge will be formalised by *quality requirements*, i.e., (reasoning) patterns that specify the behaviour of treatment selection given certain patient data. These quality requirements can be used as proof obligations in the verification of medical guidelines.

In the following section, the three types of knowledge involved (background knowledge, medical guideline, and quality requirements) are described in more detail in the context of diabetes mellitus type 2 and a formalisation in terms of temporal logic as discussed in Section 5.1 is given. In Section 8 the quality requirements are verified with the interactive theorem prover KIV.

6 Formalisation diabetes mellitus type 2 guideline

6.1 Background knowledge

In diabetes mellitus type 2 various metabolic control mechanisms are deranged and many different organ systems may be affected. Glucose level control, however, is the most important mechanism. At some stage in the natural history of diabetes mellitus type 2, the level of glucose in the blood is too high (hyperglycaemia) due to decreased production of insulin by the B cells. Oral anti-diabetics either stimulate the B cells in producing more insulin (sulfonylurea) or inhibit the release of glucose from the liver (biguanide). Effectiveness of these oral diabetics is dependent on the condition of the B cells. Finally, as a causal treatment, insulin can be prescribed. The mechanisms have been formalised in terms of temporal logic in previous work [21], and is shown in Figure 4.

For example, axiom (1) denotes the physiological effects of insulin treatment, i.e., administering insulin results in an increased uptake of glucose by the liver and peripheral tissues. Axiom (8) phrases under what conditions you may expect the patient to get cured, i.e., when the patient suffers from hyperglycaemia and insulin production of his B cells are exhausted, an increased uptake of glucose by the liver and peripheral tissues results in the patient condition changing to normoglycaemia.

(1) $\text{Drug}(\mathit{insulin}) \rightarrow \circ\, (\mathit{uptake}(\mathit{liver}, \mathit{glucose}) = \mathit{up} \wedge$
$\mathit{uptake}(\mathit{peripheral\text{-}tissues}, \mathit{glucose}) = \mathit{up})$
(2) $\mathit{uptake}(\mathit{liver}, \mathit{glucose}) = \mathit{up} \rightarrow \mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down}$
(3) $(\text{Drug}(\text{SU}) \wedge \neg \mathit{capacity}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{exhausted})$
$\rightarrow \circ\, \mathit{secretion}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{up}$
(4) $\text{Drug}(\text{BG}) \rightarrow \circ\, \mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down}$
(5) $(\circ\, \mathit{secretion}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{up} \wedge$
$\text{Condition}(\mathit{hyperglycaemia}) \wedge$
$\mathit{capacity}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{subnormal} \wedge \text{QI} \leq 27)$
$\rightarrow \circ\, \text{Condition}(\mathit{normoglycaemia})$
(6) $(\circ\, \mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down} \wedge \text{QI} > 27 \wedge$
$\mathit{capacity}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{subnormal} \wedge$
$\text{Condition}(\mathit{hyperglycaemia}))$
$\rightarrow \circ\, \text{Condition}(\mathit{normoglycaemia})$
(7) $((\circ\, \mathit{release}(\mathit{liver}, \mathit{glucose}) = \mathit{down} \vee$
$\circ\, \mathit{uptake}(\mathit{peripheral\text{-}tissues}, \mathit{glucose}) = \mathit{up}) \wedge$
$\mathit{capacity}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{nearly\text{-}exhausted} \wedge$
$\circ\, \mathit{secretion}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{up} \wedge$
$\text{Condition}(\mathit{hyperglycaemia}))$
$\rightarrow \circ\, \text{Condition}(\mathit{normoglycaemia})$
(8) $(\circ\, \mathit{uptake}(\mathit{liver}, \mathit{glucose}) = \mathit{up} \wedge$
$\circ\, \mathit{uptake}(\mathit{peripheral\text{-}tissues}, \mathit{glucose}) = \mathit{up} \wedge$
$\mathit{capacity}(\mathit{b\text{-}cells}, \mathit{insulin}) = \mathit{exhausted} \wedge$
$\text{Condition}(\mathit{hyperglycaemia}))$
$\rightarrow \circ\, (\text{Condition}(\mathit{normoglycaemia}) \vee \text{Condition}(\mathit{hypoglycaemia}))$
(9) $(\text{Condition}(\mathit{normoglycaemia}) \oplus \text{Condition}(\mathit{hypoglycaemia}) \oplus$
$\text{Condition}(\mathit{hyperglycaemia})) \wedge \neg(\text{Condition}(\mathit{normoglycaemia}) \wedge$
$\text{Condition}(\mathit{hypoglycaemia}) \wedge \text{Condition}(\mathit{hyperglycaemia}))$

Figure 4. Background knowledge $\mathcal{B}_{\text{DM2}}$ of diabetes mellitus type 2. $\text{Drug}(x)$ holds iff drug x is being administered at that moment in time. The $\oplus$ operator denotes the exclusive OR operator.

6.2 Asbru model

In Asbru, plans are hierarchically organised in which a plan refers to a number of sub-plans. The overall structure of the Asbru model of our running example (Figure 2), is shown in Figure 5. The top level plan 'Treatments_and_Control' sequentially executes the four sub-plans 'Diet', 'SU_or_BG', 'SU_and_BG', and 'Insulin_Treatments', which correspond to the four steps of the guideline fragment in Figure 2. The sub-plan 'Insulin_Treatments' is further refined by two sub-plans 'Insulin_and_Antidiabetics' and 'Insulin', which can be executed in any order.

The Asbru specifications of two plans in the hierarchy, namely 'SU_or_BG' and 'Insulin_Treatments' are defined as in Figure 6.

In the case of 'SU_or_BG' there is a relationship between the

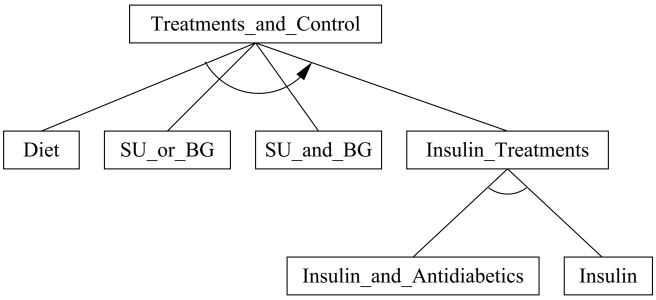


Figure 5. Asbru plan hierarchy of the diabetes mellitus type 2 guideline.

```
plan 'SU_or_BG'
  effects
    (QI ≤ 27 → SU ∈ Drugs) ∧
    (QI > 27 → BG ∈ Drugs)
  abort condition
    'condition = hyperglycaemia confirmation required'
  complete condition
    condition = hypoglycaemia ∨
    condition = normoglycaemia

plan 'Insulin_Treatments'
  body anyorder wait for one
    'Insulin_and_Antidiabetics'
    'Insulin'
```

Figure 6. Asbru specifications of two treatments recommended in the diabetes mellitus type 2 guideline.

Quetelet index (QI) and the drug administered. If the Quetelet index is less or equal than 27 then SU is administered, else BG is administered. The plan 'SU_or_BG' corresponds to step 2 in the guideline fragment of Figure 2, which completes if the patient condition improves, i.e., the patient no longer has hyperglycaemia. This is represented by the **complete condition**. The plan 'SU_or_BG' aborts when the condition of the patient does not improve, which is represented by the **abort condition**. It requires a manual confirmation to ensure that some time passes for the drugs to have an impact on the patient condition.

The plan 'Insulin_Treatments' consists of two sub-plans, which correspond to the two options of step 4 in the guideline fragment of Figure 2, i.e., either insulin is administered or insulin and antidiabetics are administered.

6.3 Quality requirements

Here, we give a formalisation of good practice medicine of medical guidelines. This extends previous work [21], which formalised good practice medicine on the basis of a theory of abductive reasoning of single treatments. The context of the formalisation given here is a fully formalised guideline, which consists, besides a number of treatments, of a control structure that uses patient information to decide on a particular treatment. This contrast with [21], which used a context of a singly chosen treatment.

Firstly, we formalise the notion of a *proper* guideline according to the theory of abductive reasoning. Let $\mathcal{B}$ be medical background knowledge, P be a patient group, N be a collection of intentions, which the physician has to achieve, and M be a medical guideline. Then M is called a *proper* guideline for a patient group P, denoted as $M \in Pr_P$, if:

(M1) $\mathcal{B} \cup M \cup P \not\models \bot$ (the guideline does not have contradictory effects), and

(M2) $\mathcal{B} \cup M \cup P \models \Diamond N$ (the guideline eventually handles all the patient problems intended to be managed)

Secondly, we formalise good practice medicine of guidelines. Let $\preceq_\varphi$ be a reflexive and transitive order denoting a preference relation with $M \preceq_\varphi M'$ meaning that M' is *at least as preferred* to M given criterion φ. With $\prec_\varphi$ we denote the order such that $M \prec_\varphi M'$ if and only if $M \preceq_\varphi M'$ and $M' \not\preceq_\varphi M$. When both $M \preceq_\varphi M'$ and $M' \preceq_\varphi M$ hold or when M and M' are incomparable w.r.t. $\preceq_\varphi$ we say that M and M' are *indifferent*, which is denoted as $M \sim M'$. If in addition to (M1) and (M2) condition (M3) holds, with

(M3) $O_\varphi(M)$ holds, where O_φ is a meta-predicate standing for an optimality criterion or combination of optimality criteria φ defined as: $O_\varphi(M) \equiv \forall M' \in \mathsf{Pr}_P : \neg(M \prec_\varphi M')$,

then the guideline is said to be *in accordance with good practice medicine* w.r.t. criterion φ and patient group P, which is denoted as $\mathsf{Good}_\varphi(M, P)$.

A typical example for O_φ is consistency of the recommended treatment order w.r.t. a preference relation $\preceq_\psi$ *over treatments*, i.e., $O_\varphi(M)$ holds if the guideline M recommends treatment T before treatment T' when $T' \prec_\psi T$ holds. For example, in diabetes mellitus type 2, a preference relation over treatments would be to minimise (1) the number of insulin injections, and (2) the number of drugs involved. This results, among others, in the following preferences: sulfonylurea drug $\sim$ biguanide drug, and insulin $\preceq_\psi$ insulin and antidiabetic $\preceq_\psi$ sulfonylurea and biguanide drug $\preceq_\psi$ sulfonylurea or biguanide drug $\preceq_\psi$ diet. A guideline M would then be in accordance with good practice medicine if it is consistent with this preference order $\preceq_\psi$, e.g., if M first recommends diet before a sulfonylurea or biguanide drug.

7 Specification in KIV

Previous sections have given the temporal logic formalisation of the background knowledge of diabetes mellitus type 2, the quality requirements, and the Asbru model of the medical guideline for diabetes mellitus type 2. In this section we discuss how these elements can be translated into KIV representations, so that they become amendable to verification.

7.1 Introduction to KIV

KIV is an integrated development environment to develop systems using formal methods [6]. The specification language of KIV is based on higher-order algebraic specifications. Reactive systems can be described in KIV by means of state-charts or parallel programs; here we use parallel programs. Parallel programs are modelled as follows. Let e denote an arbitrary (first-order) expression and v_d a dynamic variable (see below), then constructs for parallel programs include: $v_d := e$ (*assignments*), **if** ψ **then** ϕ_1 **else** ϕ_2 (*conditionals*), **while** ψ **do** ϕ (*loops*), **var** $v_d = e$ **in** ϕ (*local variables*), **patom** ϕ **end** (*atomic execution*), $\phi_1 \parallel \phi_2$ (*interleaved execution*), and $[p\#(e; v_d)]$ (*call to procedure p with value parameters e and reference parameters* v_d). The semantics of this extended language is defined in [1].

The correctness of systems is ensured by constructing proofs in an interactive theorem prover which is based on higher order logic with special support for temporal logic, i.e., future-time linear temporal logic [4]. The logic of Table 1 is extended with static variables v_s, which are variables that are mapped to the same element in the universe of discourse at each time point. Dynamic variables v_d, such as program variables, may have different interpretations at different time points. In the upcoming sections, the use of static variables will be explicitly mentioned. A speciality of KIV is the use of primed and double-primed variables: a primed variable v_d' represents the value of this variable after a system transition, the double-primed variable v_d'' is interpreted as the value after an environment transition. System and environment transitions alternate, with v_d'' being equal to v_d in the successive state (cf. Figure 7 and Section 8.1).

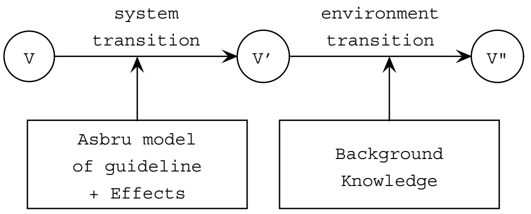


Figure 7. The relation between unprimed and primed variables as two distinct transitions: the system transition (including the Asbru model and its effects) and the environment transition (including the background knowledge).

7.2 Specification methodology in KIV

The guideline and patient can be looked upon as a system (guideline) that interacts with the environment (patient). KIV allows a clear distinction between system and environment transitions by using primed and double-primed variables. Therefore, the Asbru model is only allowed to map variables into primed variables, whereas the environment is only allowed to map primed variables into double primed variables. System and environment transitions alternate (Figure 7).

However, system transitions in Asbru may involve a large number of steps (e.g., signals, plan state changes) before the model reaches a stable state from which no further step can be made unless time progresses or the environment changes. Asbru is mainly a control oriented language and many control steps are not considered to take any real time at all. In an interactive theorem prover like KIV, this behaviour can be modelled by the introduction of two transition types, *micro-steps* and *macro-steps* [36]. Micro-steps are technical Asbru steps where time and environment are not allowed to change. Macro-steps are temporal steps in which interaction can occur with the environment (e.g., plan activations) and are only executed when there are no micro-steps possible. The variable 'Tick', controlled by the symbolic execution of the Asbru semantics, holds when a macro-step occurs.

In KIV, system descriptions are represented by means of a set of algebraic specifications. These algebraic specifications can be enriched with additional algebraic structures, which form a dependency structure between the different specifications. To maximise re-usability, several layers are used for representing our framework in KIV. The lowest layer in this dependency structure consists of standard data structures like Booleans and sets, which are typically obtained from libraries in KIV. On top of that, all data structures are represented necessary for representing the semantics of Asbru. The remaining layers consist of the structures dependent on the specific guideline under study. On top of the standard data structures, additional data structures are represented. For the diabetes case study, the data types are modelled as enumeration types. On top of the asbru semantics and data structures the background knowledge is represented. The top layer consists of the control structure of the guideline, which is the structure of Figure 5 in the diabetes case study (cf. Figure 8).

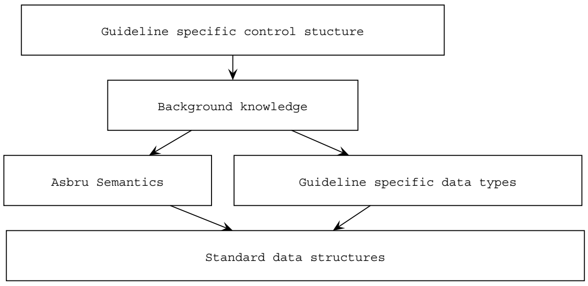

Guideline specific control stucture
Background knowledge
Asbru Semantics
Guideline specific data types
Standard data structures

Figure 8. Dependency structure of Asbru specifications with $A \rightarrow B$ denoting that A depends on B

7.3 Specification of background knowledge in KIV

The background knowledge is translated into algebraic specifications in KIV. All background knowledge axioms have been reformulated in terms of preconditions and postconditions. Every element that refers to the current point in time is interpreted as a precondition and each element that refers to the next point in time is interpreted as a postcondition. The values of these elements are stored in a data structure, denoted by 'Patient'. The patient is modelled by a sequence of pairs $[v, c]$, where v is the name of a variable and c a constant denoting the value of that variable, depending on the point in time. Updates to the patient record are done by appending a pair to the end of the sequence. Moreover, the most recent value of a variable v in a sequence s is given by the term $s[v]$. An example of the final translation can be found in Figure 9.

predicates
  Knowledge : *patient* × *patient*;
axioms
BDM2-1:
  Knowledge(*pre*, *post*) → (*insulin* ∈ *pre*[*treatment*] →
    post[*uptake*(*liver*,*glucose*)] = *up* ∧
    post[*uptake*(*peripheral-tissues*, *glucose*)] = *up*)
BDM2-8:
  Knowledge(*pre*, *post*) → (*post*[*uptake*(*liver*,*glucose*)] = *up*
    ∧ *post*[*uptake*(*peripheral-tissues*,*glucose*)] = *up*)
    ∧ *pre*[*capacity*(*b-cells*,*insulin*)] = *exhausted*
    ∧ *pre*[*condition*] = *hyperglycaemia* →
    post[*condition*] = *normoglycaemia*)

Figure 9. Background knowledge in KIV as a first order predicate using pre- and postconditions, i.e., *pre* and *post* are shorthand notations for patient data structures with *pre*[v] = c and *post*[v] = c referring to the condition $v = c$ of the patient in the current and next state respectively. The use of pre and $post$ variables is necessary to parameterise the background knowledge for arbitrary patient data structures. In addition, two translated rules from the background formalisation in [21] are shown with BDM2-i representing Axiom (i) (cf. Figure 4).

7.4 Specification of Asbru in KIV

As each Asbru plan has a strict format, an algebraic function 'mk-asbru-def' has been defined for the translation of Asbru plans into

KIV specifications. By calling ‘mk-asbru-def’ with the parameters that constitute a plan, translation of any guideline in Asbru becomes straightforward. The parameters consist of the various conditions that control plan state changes, the control type of sub-plans, a list of sub-plans, a retry value (for aborted plans), a wait-for condition (for mandatory sub-plans), and an optional wait-for flag (whether to wait for sub-plans). As there are quite a number of parameters, default values are provided to ease specification.

The Asbru semantics is implemented as a parallel program, parametrised with a given Asbru model. Temporal properties of this program are proven using symbolic execution and induction [1].

7.5 Specification of quality requirements in KIV

With the help of KIV, we have verified that the diabetes guideline is proper, i.e., that the guideline satisfies conditions (M1) and (M2) as defined in Section 6.3, which is discussed in detail in Subsections 8.1 and 8.2. Meta-level quality requirements are verified in KIV using a sequent $\Gamma \vdash \Sigma$ where the succedent Σ is some instantiation of (M3) and the antecedent Γ is a fixed structure that consists of the initial state of the patient and the Asbru model, the Asbru model, the effects of treatments, the background knowledge, and the environment assumptions. The sequent in Figure 10 is an example specification in KIV of the quality requirement that each patient is eventually cured from hyperglycaemia.

/ Initial state of patient */*
$\text{Patient}[\mathit{condition}] = \mathit{hyperglycaemia}$,
/ Initial state of guideline */*
$\text{AS}[\text{Treatments\_and\_Control}] = \mathit{inactive}, \ldots,$
/ Asbru model */*
$[\mathit{asbru\#}(\text{Treatments\_and\_Control}; \text{AS}, P)]$,
/ Effects */*
$\Box\ (\text{AS}[\text{SU\_or\_BG}] = \mathit{activated} \leftrightarrow$
$\quad \text{BG} \in \text{Patient}'[\mathit{treatment}] \wedge \ldots)$,
/ Background knowledge */*
$\Box\ \text{Knowledge}(\text{Patient}', \text{Patient}'')$
/ Environment assumption */*
$\Box\ (\text{AS}''[\text{Treatments\_and\_Control}] =$
$\quad \text{AS}'[\text{Treatments\_and\_Control}] \wedge \ldots)$
$\vdash$
/ Property */*
$\Diamond\ (\text{Patient}[\mathit{condition}] = \mathit{hypoglycaemia} \vee$
$\quad \text{Patient}[\mathit{condition}] = \mathit{normoglycaemia})$

Figure 10. Specification in KIV of the quality requirement that each patient is eventually cured from hyperglycaemia.

The *initial state* of the *patient* and the *Asbru model* are represented using additional data structures [35]. The patient data is represented in a data structure ‘patient-data-history’, which in Figure 10 is set to the patient group $\{\text{Condition}(\mathit{hyperglycaemia})\}$. The initial state of the Asbru model is represented using a data structure ‘AS’ of type ‘asbru-state’, which keeps track of all plan states over time, and in which initially each plan is set to inactive. The *Asbru model* of the guideline describes the control structure, and its specification in KIV has already been discussed in Section 7.4. The *effects of treatments* specify in KIV the behaviour of plans in the Asbru model. This is a direct translation of the **effects** attribute used in the Asbru model, which specifies the expected behaviour of plans (cf. Section 6.2). In our diabetes case study the effects of plans are the administration of a certain drug as soon as the plan becomes activated, which may depend on the value of other variables like the Quetelet index (cf. Section 6.2). The *background knowledge* is represented in the sequent using the first-order predicate ‘Knowledge’ and has already been discussed in Section 7.3. The environment is in principle allowed to change every variable arbitrarily. The *environment assumptions* restrict the behaviour of the environment. These restrictions (1) forbid the environment to change some variable, (2) force the environment to deterministically change a variable (e.g., advancing a clock), and (3) guarantee certain variable assignments in a nondeterministic way (e.g., the existence of a value when a signal is sent).

8 Verification using KIV

8.1 Consistency of background knowledge

Property (M1) ensures that the formal model including the Asbru guideline and the background knowledge is consistent. The initial state is – in our case – described as a set of equations and it has been trivial to see that they are consistent. The guideline is given as an Asbru plan. The semantics of any Asbru plan is defined in a programming language where every program construct ensures that the resulting reactive system is consistent: in every step, the program either terminates or calculates a consistent output for arbitrary input values. The Asbru plan, thus, defines a total function between unprimed and primed variables in every step (Figure 7). The formula defining the effects maps the output variables of the guideline to input variables of the patient model. Again, it has been trivial to see that this mapping is consistent.

The background knowledge defines our patient model. We consider the patient to be part of the environment which is the relation between the primed and the double primed variables in every step. If the patient model ensures that for an arbitrary primed state there exists a double primed state, the overall system of alternating guideline and environment transitions is consistent: given an initial (unprimed) state, the guideline calculates an output (primed) state; the effects define a link between the variables of the guideline and the variables of the patient model; the patient model reacts to the (primed) output state and gives a (double primed) state which is again input to the Asbru guideline in the next step. In other words, the relation between the unprimed and the double primed state is the complete state transition. The additional environment assumptions referring to the Asbru environment do not destroy consistency as the set of restricted variables of the environment assumption is disjunct to the set of variables of the patient model.

It remains to ensure consistency of the background knowledge which we defined as a predicate ‘knowledge’. Consistency can be shown by proving the property

$$\forall pre.\ \exists post.\ \text{‘knowledge’}(pre, post)$$

which ensures that the relation is total. In order to prove that this property holds an example patient has been constructed. Verifying that the example patient is a model of the background knowledge has been fully automatic.

8.2 Successful treatment

In order to verify property (M2), i.e., the guideline eventually manages to control the glucose level in the patient’s blood, a proof has been constructed. The verification strategy in KIV is symbolic execution with induction [1]. The plan state model introduced in [3] defines the semantics of the different conditions of a plan and is implemented in KIV by a procedure called 'asbru', which is symbolically executed. Each plan can be in a certain state, modelled with a variable 'AS' (i.e., 'inactive', 'considered', 'ready', 'activated', and 'aborted' (or 'completed')) and a transition to another state depends on its conditions. In the initial state, the top level plan 'Treatments_and_Control' (abbreviated 'tc') is in 'inactive' state. After executing the first step, the plan is 'considered', after which execution continues as described in [3]. The execution is visualised in a proof tree (cf. Figure 11), where the bottom node is the start of the execution and splits if there is a case distinction.

Patients whose capacity of the B cells is 'normal' are cured with diet, while for other patients diet will not be sufficient. In this case, we assume that the doctor eventually aborts the diet treatment. We use induction to reason about the unspecified time period in which diet is applied. As an invariant,

$$Patient[\text{'}\textit{capacity(B-cells,insulin)}\text{'}] \neq normal$$

is used. In the next step, the doctor has either aborted 'diet' or 'diet' is still active. In the second case, induction can be applied. When 'diet' is aborted, 'tc' sequentially executes the next plan, which is 'SU_or_BG' (cf. Figure 5).

The second treatment 'SU_or_BG' goes, as each Asbru plan, through a sequence of states, i.e., 'inactive', 'considered', 'ready', 'activated', and 'aborted', and thus becomes first 'considered' and after some steps becomes 'activated' (cf. Figure 11). In this case, either SU or BG is prescribed, depending on the Quetelet index QI. For a patient whose B cell capacity is 'subnormal', the background knowledge ensures that the condition of the patient improves. Thus, for the rest of the proof we can additionally assume that

$$Patient[\text{'}\textit{capacity(B-cells,insulin)}\text{'}] \neq subnormal$$

After 'SU_or_BG' aborts, the third treatment ('SU_and_BG') is executed in similar fashion, where patients with nearly exhausted B cell capacity are cured. Thus, after aborting the first three treatments the precondition concerning the B cell capacity can be strengthened to

$$\begin{aligned} & Patient[\text{'}\textit{capacity(B-cells,insulin)}\text{'}] \neq \text{'}normal\text{'} \\ \wedge\ & Patient[\text{'}\textit{capacity(B-cells,insulin)}\text{'}] \neq \text{'}subnormal\text{'} \\ \wedge\ & Patient[\text{'}\textit{capacity(B-cells,insulin)}\text{'}] \neq \text{'}nearly\text{-}exhausted\text{'} \end{aligned}$$

which, under the assumption that the only possible values of the capacity are normal, subnormal, nearly-exhausted, and exhausted, yields:

$$Patient[\text{'}\textit{capacity(B-cells,insulin)}\text{'}] = exhausted$$

This statement together with the background knowledge ensures that the prescription of insulin, which is prescribed in both final treatments 'Insulin' and 'Insulin_and_Antidiabetics', finally cures the patient.

8.3 Optimality of treatment

With respect to property (M3), an optimality criterion of the guideline is that no treatments are prescribed that are not in accordance with good practice medicine (Section 6.3), i.e., some preference relation $\preceq$ between treatments exists and the guideline never prescribes a treatment T such that $T \preceq T'$ and T' cures the patient group under consideration.

In our case study the preference for treatments is based on the minimisation of (1) the number of insulin injections, and (2) the number

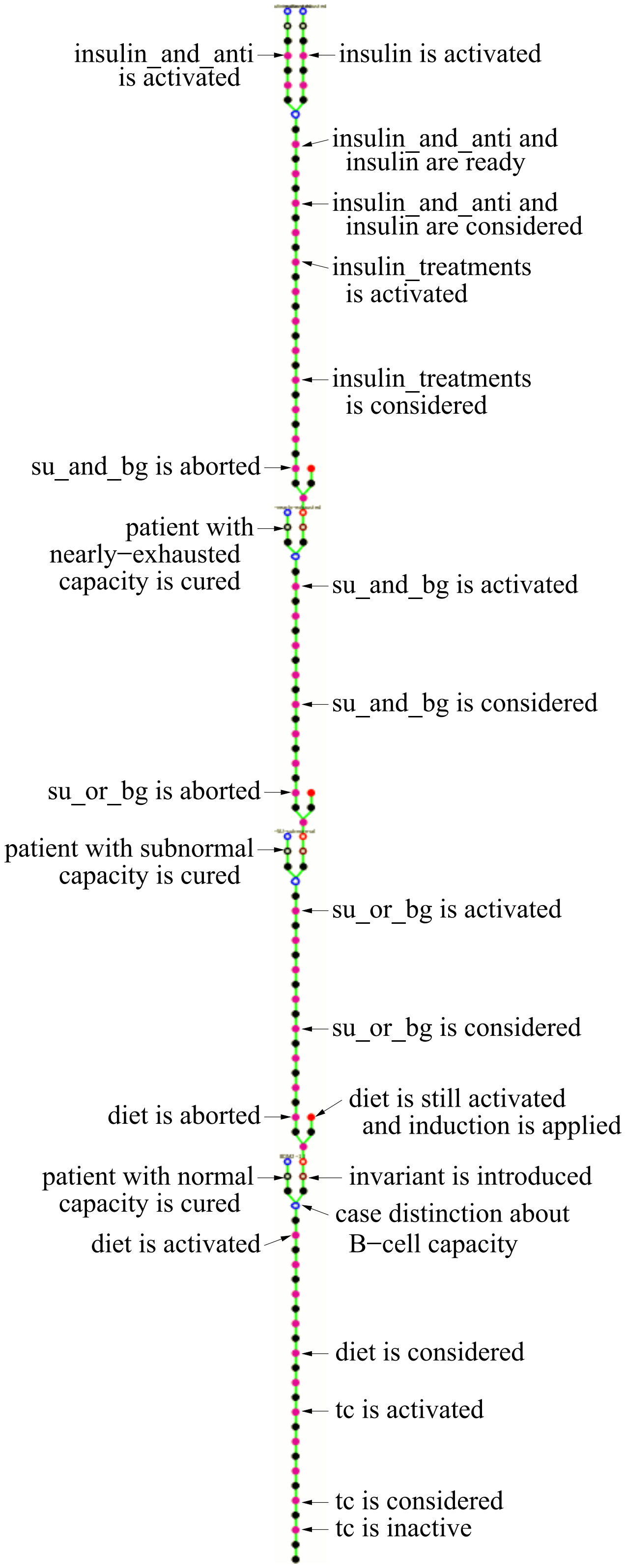

insulin_and_anti is activated
insulin is activated
insulin_and_anti and insulin are ready
insulin_and_anti and insulin are considered
insulin_treatments is activated
insulin_treatments is considered
su_and_bg is aborted
patient with nearly−exhausted capacity is cured
su_and_bg is activated
su_and_bg is considered
su_or_bg is aborted
patient with subnormal capacity is cured
su_or_bg is activated
su_or_bg is considered
diet is aborted
diet is still activated and induction is applied
patient with normal capacity is cured
invariant is introduced
case distinction about B−cell capacity
diet is activated
diet is considered
tc is activated
tc is considered
tc is inactive

Figure 11. Overview of the proof that the guideline eventually manages all patient problems, which is explained in Section 8.2.

of drugs involved (cf. Section 6.3). We have defined this using a reflexive, transitive order $\leq$ such that for all treatments T, it holds that $insulin \leq T$ and $T \leq diet$. Furthermore, the treatments prescribing the oral anti-diabetics sulfonylurea and biguanide are incomparable. The proof obligation is then as follows:

$$\Box(\forall_T\colon Good_{\leq}(T, Patient) \rightarrow T \leq Patient[\textit{`treatment'}])$$

where $Good_{\leq}(T, Patient)$ denotes that T is a treatment according to good practice medicine for $Patient$, as defined in [24]. To prove this, the following axiom was added to the system:

$$\Box Patient[\textit{`QI'}] = Patient''[\textit{`QI'}]$$

i.e., the Quetelet index does not change during the run of the protocol. This axiom is needed, because the decision of prescribing a treatment is not exactly at the same time as the application of the treatment and therefore the decision of prescribing this treatment could be based on a patient with a different Quetelet index than the patient that actually takes the drugs.

Proving this property in KIV was done in approximately 1 day using several heuristics for the straightforward parts. The theorem was proven using two lemmas for two specific patient groups. In total, it took approximately 500 steps, of which nearly 90% were done automatically, to verify this property.

8.4 Order of treatments

Finally, another instance of (M3) was proven. This property phrases that the order of any two treatments in the protocol is consistent with the order relation as we have defined in Subsection 6.3. In other words, in case a patient may receive multiple treatments, the less radical treatments are tried first. The formalisation of the property in KIV was done as follows:

$$\begin{aligned}&\Box\forall_T(Tick \wedge T = Patient[\textit{`treatment'}]\\&\quad\rightarrow \Box(\textbf{last} \vee (Tick \rightarrow \neg(T \leq Patient[\textit{`treatment'}]))))\end{aligned}$$

At each time, the current treatment is bound to a static variable (i.e., unchanged by symbolic execution) T, which can be used to compare against subsequent steps in the protocol. For any future steps, we require that either the protocol completes (**last** holds) or that activated treatments are not more preferred than T. The additional 'Tick' variable is needed in the formalisation to abstract from technical system steps.

This property also had a high degree of automation with roughly 800 steps in total. The reason for this slightly higher number of steps is due to nested temporal operators.

9 Discussion

As the interest in medical guidelines continues to grow, there is a need for criteria to asses the quality of medical guidelines. An important method for the appraisal of medical guidelines was introduced by the AGREE collaboration [9]. A solid foundation for the application of *formal methods* to the quality checking of medical guidelines, using simulation of the guideline [15, 31] and theorem proving techniques [25], can also be found in literature.

In [25], logical methods have been used to analyse properties of guidelines, formalised as task networks. In [24], it was shown that the theory of abductive diagnosis can be taken as a foundation for the formalisation of quality requirements of a medical guideline in temporal logic. This result has been used to verify quality requirements of good practice medicine of treatments [21]. However, in the latter work, the order between treatment depending on the condition of the patient and previous treatments was ignored. In this paper, we consider elements from both approaches by including medical background knowledge in the verification of complete networks of tasks. This required a major change to the previous work with respect to the formulation of quality criteria, because quality is now defined with respect to a complete network of tasks instead of individual treatments as presented in [24].

Compared to previous work concerning the verification of networks of tasks, the meta-level approach we have presented here has a number of advantages. In the meta-level approach, quality is defined independently of domain specific knowledge, and, consequently, proof obligations do not have to be extracted from external sources. One successful attempt of the latter was reported in [18], where quality criteria are formalised on the basis of instruments to monitor the quality of care in practice, i.e., medical indicators. Firstly, the question is whether these indicators, based on compliance with medical guidelines, coincide with the quality of the guideline itself. Secondly, it has been our experience that it is far from easy to find suitable properties in external sources, because these sources may not be completely applicable, e.g., typically, other guidelines may address different problem in the management of the same disease. Thirdly, many useful quality criteria of guidelines are implicit, making this approach fundamentally limiting. In this sense, the meta-level approach provides a more systematic method for the formulation of proof obligations and, thus, verification of medical guidelines.

In summary, in this study we have setup a general framework for the verification of medical guidelines, consisting of a medical guideline, medical background knowledge, and quality requirements. A model for the background knowledge of glucose level control in diabetes mellitus type 2 patients was developed based on a general temporal logic formalisation of (patho)physiological mechanisms and treatment information. Furthermore, we developed a theory for quality requirements of good practice medicine based on the theory of abductive diagnosis. This model of background knowledge and theory of quality requirements were then used in a case study in which we verified several quality criteria of the diabetes mellitus type 2 guideline used by the Dutch general practitioners. In the case study we use Asbru to model the guideline as a network of tasks and KIV for the formal verification.

In the course of our study we have shown that the general framework that we have setup for the formal verification of medical guidelines with medical background knowledge is feasible and that the actual verification of the proposed quality criteria can be done with a high degree of automation. We believe both the inclusion of medical background knowledge and task networks to be necessary elements for adequately supporting the development and management of medical guidelines.

10 Comparison with other formal verification techniques

Formal methods: Verification using symbolic calculation can be mechanised using the methods of several types of reasoning, such as model checking, constraint solving, theorem proving, etc. Figure 12 shows a range of formal methods ranging from cheap to incomplete to very expensive and complete (loosely based on a picture by Rushby). The work that is presented in this paper is of the latter kind, which has certain advantages, e.g., it provides insight in the proof structure. For each case, it is relatively easy to inspect the proof tree and to find out the reason why a certain quality criterion holds. On the other hand, KIV is a tool with a very expressive logic, which may result in an additional overhead when verifying quality criteria of medical guidelines. Thus, it makes sense to look at cheaper methods for verification of medical guidelines. This is particularly important when guidelines are rapidly updated, where fully automated formal methods are most realistic. Below, work on model checking and automated theorem proving of medical guidelines is briefly discussed.

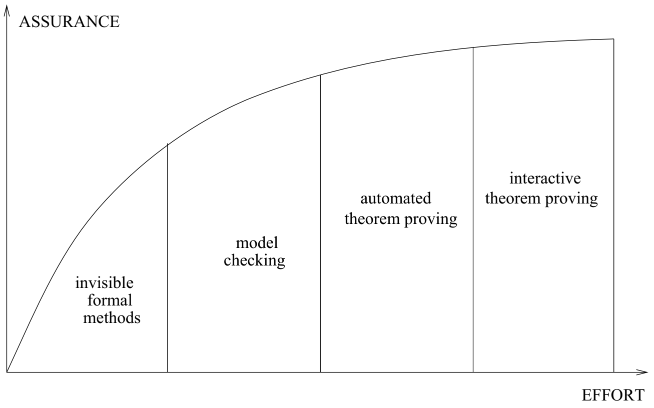


Figure 12. A spectrum of formal methods for formal verification allowing a tradeoff in the properties one can verify (assurance dimension) against the effort one needs to invest to obtain results (effort dimension).

Model checking: Model checking is an effective technique for verifying properties of a formal system. In model checking, a specification about a model, which is usually some form of transition system, is expressed as (temporal) logic formulas, and efficient algorithms traverse the states of the system to verify whether the specification holds or not. Extremely large state-spaces can be traversed in a short amount of time. The first model checkers verified the correctness of discrete state systems, but have been extended to also deal with real-time and probabilistic reasoning.

In the Protocure project, a mapping has been developed for automatically transforming guidelines in the Asbru language into SMV for model checking purposes [7]. As the mapping is made into SMV, this transformation abstracts from the notion of time. Hence, not every property can be verified using SMV [26]. Model checking has been found to be very useful when constructing the Asbru model. [12] defines a number of structural properties which should be fulfilled by a good quality Asbru model. By model checking these structural properties of the Asbru model, one can quickly check the model during development. Hence, model checking provides a good trade-off between effort and assurance for these kind of properties, however, the framework as specified in [7] is unable to deal with more complex properties that deal for example with time.

In another study [19], model checking has been used to check the conformance of medical guidelines with medical protocols, which are local adaptations by hospitals of medical guidelines. A different view towards medical guidelines was followed in [19] compared to the program-like view presented in the current paper. As medical guidelines often omit many details, e.g., common sense reasoning about first informing a patient before treatment, guidelines are often under-constrained. In [19] a constraint-based approach is used for model checking the conformance of medical protocols. Additional background knowledge can be incorporated in the model checking approach by using modular model checking [22]. This allows one to verify a property with respect to a restricted part of the model. For example, one can restrict the model to those states that adhere to common sense medical practice, such as the fact that diagnosis usually occurs before treatment of the patient.

Automated theorem proving: Previously, it was shown that for reasoning about models of medical knowledge, for example in the context of medical expert systems [23], classical automated reasoning techniques (e.g., [33, 46]) are a practical option. In [20], we studied the use of automatic theorem proving techniques for quality checking medical guidelines. In this context, reasoning about Asbru plans is not feasible, however, simple treatment plans can be encoded directly in temporal logic. Translation of temporal logic yields a restricted first-order theory, e.g., the temporal formula $\mathbf{G}p$ can be interpreted as by $\forall_{t'} : (t \leq t' \rightarrow p)$. Such a formalisation is suitable for use in standard resolution-based theorem provers. Note that in practice, this is not a fully automated process, as the theorem prover needs to be guided in the use of (resolution-)strategies and sometimes it is helpful to define lemmas. Nonetheless, automated theorem provers require less interaction than interactive theorem provers. Furthermore, it is possible to add background knowledge to the system, whereas, adding background knowledge to a transition system will generally result in a state explosion making model checking infeasible.

ACKNOWLEDGEMENTS

We would like to thank all members of the Protocure project for providing a stimulating research environment.

REFERENCES

[1] M. Balser, *Verifying Concurrent Systems with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction*, Ph.D. dissertation, University of Augsburg, Augsburg, Germany, 2005.

[2] M. Balser, O. Coltell, J. van Croonenborg, C. Duelli, F. van Harmelen, A. Jovell, P. Lucas, M. Marcos, Misch. S., W. Reif, K. Rosenbrand, A. Seyfang, and A. ten Teije, 'Protocure: Supporting the development of medical protocols through formal methods', in *Computer-Based Support for Clinical Guidelines and Protocols*, eds., K. Kaiser, S. Miksch, and S. Tu, pp. 103–107. IOS Press, (2004).

[3] M. Balser, C. Duelli, and W. Reif, 'Formal semantics of Asbru - an overview', in *Proceedings of the International Conference on Integrated Design and Process Technology*, Passadena, (2002). Society for Design and Process Science.

[4] M. Balser, C. Duelli, W. Reif, and G. Schellhorn, 'Verifying concurrent systems with symbolic execution', *Journal of Logic and Computation*, **12**(4), 549–560, (2002).

[5] M. Balser, C. Duelli, W. Reif, and J. Schmitt, 'Formal semantics of asbru – v2.12', Technical report, University of Augsburg, (June 2006). Url: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

[6] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, 'Formal system development with KIV', in *Fundamental Approaches to Software Engineering*, ed., T. Maibaum, number 1783 in LNCS. Springer-Verlag, (2000).

[7] S. Bäumler, M. Balser, A. Dunets, W. Reif, and J. Schmitt, 'Verification of medical guidelines by model checking – a case study', in *Proceedings of 13th International SPIN Workshop on Model Checking of Software*, ed., A. Valmari, volume 3925 of *LNCS*, pp. 219–233. Springer-Verlag, (2006).

[8] P. Clayton and G. Hripsak, 'Decision support in healthcare', *International Journal of Biomedical Computing*, **39**, 59–66, (1995).

[9] AGREE Collaboration, 'Development and validation of an international appraisal instrument for assessing the quality of clinical practice guidelines: the agree project', *Qual Saf Health Car*, **12**, 18–23, (2003).

[10] P.A. de Clercq, J.A. Blom, H.H.M. Korsten, and A. Hasman, 'Approaches for creating computer-interpretable guidelines that facilitate decision support', *Artificial Intelligence in Medicine*, **31**(1), 1–27, (2004).

[11] D. Dickenson and P. Vineis, 'Evidence-based medicine and quality of care', *Health Care Analysis*, **10**, 243–259, (2002).

[12] G. Duftschmid and S. Miksch, 'Knowledge-based verification of clinical guidelines by detection of anomalies', *OEGAI Journal*, 37–39, (1999).

[13] E. Allen Emerson, 'Temporal and modal logic.', in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, 995–1072, (1990).

[14] *Clinical Practice Guidelines: Directions for a New Program*, eds., M. Field and K. Lohr, National Academy Press, Institute of Medicine, Washington D.C., 1990.

[15] J. Fox and S. Das, *Safe and Sound: Artificial Intelligence in Hazardous Applications*, AAAI Press, 2000.

[16] J. Fox, N. Johns, A. Rahmanzadeh, and R. Thomson, 'PRO*forma*: A method and language for specifying clinical guidelines and protocols', in *Medical Informatics Europe*, eds., J. Brender, J.P. Christensen, Scherrer. J.R., and P. McNair, pp. 516–520, (1996).

[17] J. Fox, N. Johns, A. Rahmanzadeh, and R. Thomson, 'PRO*forma*: a general technology for clinical decision support systems', *Computer Methods and Programs in Biomedicine*, **54**, 59–67, (1997).

[18] M. van Gendt, A. van Teije, R. Serban, and F. van Harmelen, 'Formalising medical quality indicators to improve guidelines', in *AIME*, number 3581 in LNAI, pp. 201–220. Springer Verlag, (2005).

[19] A. Hommersom, P. Groot, and P. Lucas, 'Checking guideline conformance of medical protocols using modular model checking', in *The 18th Belgium-Netherlands Conference on Artificial Intelligence*, pp. 173–180, (2006).

[20] A. J. Hommersom, P. J. F. Lucas, and P. van Bommel, 'Automated theorem proving for quality-checking medical guidelines', in *Proceedings of CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR)*, (2005).

[21] A.J. Hommersom, P.J.F. Lucas, and M. Balser, 'Meta-level Verification of the Quality of Medical Guidelines Using Interactive Theorem Proving', in *Logics in Artificial Intelligence: 9th European Conference*, volume 3229 of *Lecture Notes in Computer Science*, pp. 654–666, Lisbon, Portugal, (September 2004). Springer-Verlag.

[22] O. Kupferman and M.Y. Vardi, 'Modular model checking', *Lecture Notes in Computer Science*, **1536**, 381–401, (1998).

[23] P. J. F. Lucas, 'The Representation of Medical Reasoning Models in Resolution-based Theorem Provers', *Artificial Intelligence in Medicine*, **5**, 395–419, (1993).

[24] P.J.F. Lucas, 'Quality checking of medical guidelines through logical abduction', in *Proceedings of AI-2003, the 23rd SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, eds., F. Coenen, A. Preece, and A.L. Mackintosh, volume XX, pp. 309–321, London, (2003). Springer.

[25] M. Marcos, M. Balser, A. ten Teije, and F. van Harmelen, 'From informal knowledge to formal logic: A realistic case study in medical protocols', in *Proceedings of EKAW*, pp. 49–64. Springer, (2002).

[26] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[27] S. Miksch, 'Plan management in the medical domain', *AI Communications*, **12**(4), 209–235, (1999).

[28] M. Peleg, A. Boxwala, O. Ogunyemi, P. Zeng, S. Tu, R. Lacson, E. Begnstam, and N. Ash, 'GLIF3: The evolution of a guideline representation format', in *Proc. AMIA Annual Symposium*, pp. 645–649, (2000).

[29] M. Peleg, L.A. Gutnik, V. Snow, and V.L. Patel, 'Interpreting procedures from descriptive guidelines', *Journal of Biomedical Informatics*, **39**(2), 184–95, (2006).

[30] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R.A. Greenes, R. Hall, P.D. Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E.H. Shortliffe, and M. Stefanelli, 'Comparing computer-interpretable guideline models: a case-study approach', *Journal of the American Medical Informatics Association*, **10**(1), 52–68, (2003).

[31] S. Quaglini, M. Stefanelli, A. Cavallini, G Micieli, C. Fassino, and C. Mossa, 'Guideline-based careflow system', *Artificial Intelligence in Medicine*, **20**(1), 5–22, (2000).

[32] R. Reiter, 'Equality and domain closure in first order databases', *Journal of ACM*, **27**, 235–249, (1980).

[33] J. A. Robinson, 'Automated Deduction with Hyperresolution', *International Journal of Computatational Mathematics*, **1**, 23–41, (1965).

[34] G.E.H.M. Rutten, S. Verhoeven, R.J. Heine, W.J.C. de Grauw, P.V.M. Cromme, and K. Reenders, 'NHG-standaard diabetes mellitus type 2 (eerste herziening)', *Huisarts Wet*, **42**, 67–84, (1999).

[35] J. Schmitt, M. Balser, and W. Reif, 'Complementary material to Deliverable D4.2b: Improved Verification System', in *Protocure II - Integrating formal methods in the development process of medical guidelines and protocols*, (2005).

[36] J. Schmitt, M. Balser, and W. Reif, 'Support for Interactive Verification of Asbru in KIV', Technical Report 2006-16, Universität Augsburg, Institut für Informatik, (June 2006).

[37] A. Seyfang, R. Kosara, and S. Misch, 'Asbru's reference manual, asbru version 7.3', Technical Report Asgaard-TR-20002-1, Vienna University of Technology, Institute of Software Technology, (2002).

[38] A. Seyfang, S. Miksch, P. Votruba, K. Rosenbrand, J. Wittenberg, J. von Croonenborg, W. Reif, M. Balser, J. Schmitt, T. van der Weide, P. Lucas, and A. Hommersom, 'D2.2a Specification of Formats of Intermediate, Asbru and KIV Representations', in *Protocure II - Integrating formal methods in the development process of medical guidelines and protocols*, (2004).

[39] A. Seyfang and J. Schmitt, 'D2.3b Asbru-to-KIV translator', in *Protocure II - Integrating formal methods in the development process of medical guidelines and protocols*, (2004).

[40] Y. Shahar, S. Miksch, and P. Johnson, 'The asgaard project: A task-specific framework for the application and critiquing of time-orientied clinical guidelines', *Artificial Intelligence in Medicine*, **14**, 29–51, (1998).

[41] S.E. Strauss, W.S. Richardson, P. Glasziou, and R.B. Haynes, *Evidence-based Medicine - How to Practice and Teach EBM*, Churchill Livingstone, 2005.

[42] S. Tu and M. Musen, 'A flexible approach to guideline modeling', in *Proceedings of American Medical Informatics Association Symposium (AMIA 1999)*, pp. 420–424, (1999).

[43] S. Tu and M. Musen, 'From guideline modeling to guideline execution: Defining guideline based decision-support services', in *Proceedings of American Medical Informatics Association Symposium*, pp. 863–867, Los Angeles, CA, (1999).

[44] S. Woolf, R. Grol, A. Hutchinson, M. Eccles, and J. Grimshaw, 'Potential benefits, limitations, and harms of clinical guidelines', *British Medical Journal*, **318**, 527–530, (1999).

[45] S.H. Woolf, 'Evidence-based medicine and practice guidelines: an overview', *Cancer Control*, **7**, 362–367, (2000).

[46] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

Appendix A - Specification of Asbru in KIV

This appendix gives a bit more details about the specification of and reasoning about Asbru plans in KIV. More details about the representation is described in Protocure deliverables [38, 39] and the technical report [36].

The syntax of Asbru is defined with several algebraic specifications in KIV. Figure 13 gives an overview of the specifications and their dependency structure. The specifications with a box 'CUT' attached belong to the library specifications included in KIV and are not shown in detail. We discuss only some of the more important design choices in more detail below.

The 'asbru-clock-basic' specification defines the data type 'asbru-clock', which is a two-component counter, with the first component being either an integer or infinity, and the second component being a natural number. The first counter of the clock counts the time steps the system has gone through, i.e., the macro-steps (cf. Section 7.2). An integer is used as the absolute number is unimportant. This allows lemmas to be inserted at different time points without the difficulty with natural numbers that there exists some zero time point such that

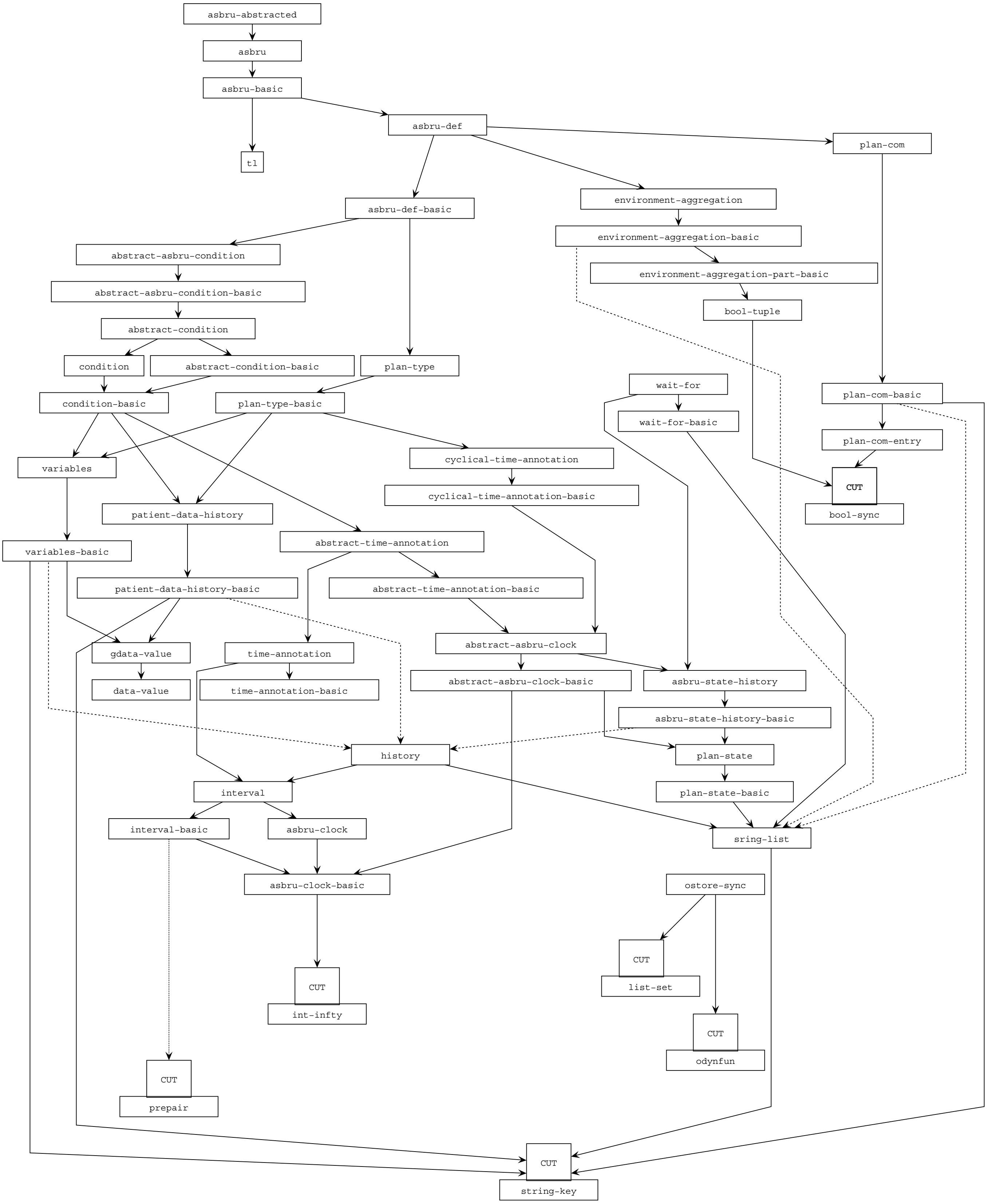


Figure 13. Definition of syntax of Asbru plans.

one cannot go back infinitely back in time. The second counter is a micro-step counter.

The ‘asbru-clock’ enriches the asbru clock, which adds functionality for moving back and forward in time. As micro-steps are technical steps that do not represent real time steps they are not related to concepts such as ‘earlier’ or ‘later’. It is therefore not possible to address individual micro-steps, but only to a list of states that has been reached in between two macro-steps.

The ‘interval-basic’ creates a rudimentary time-interval using a pair of asbru-clocks.

The ‘ostore-sync’ specification adds the specification of the predicate ‘sync’. This is needed to come around difficulties with concurrent access to data types within synchronous parallel execution. In general synchronous write access from more than one process to one variable is seen as a clash and the result of such a clash can be defined in a number of ways. For example, the result can be (1) chosen from the result of one of the processes, (2) arbitrary, (3) the results of both processes (e.g., when they access different fields in an array), or (4) an inconsistency leading to an abort of the program. The ‘sync’ predicate postpones the decision how to react to clashes and allows it to be specified on the case study level.

The ‘history’ specification is a generic specification with the type of the included dynamic function left undefined. This allows one to define generic simplification rules and reuse them for multiple specifications. In the Asbru specification the history construct is used for the variable history, the Asbru state history, and the patient data history. The selectors in the history are basically time points, but intervals have also been added to increase modularisation.

The most important data structures within the specification of Asbru are the ‘asbru state’, ‘patient data’, and ‘patient’. The ‘asbru state’ stores all configurations of Asbru plans, i.e., their current state according to the semantics of the state-chart (cf. Figure 3). The ‘patient data’ stores all the known values about the patient. Note, that there is a difference between the ‘patient’ data structure and ‘patient data’ data structure, as the former contains information about the *actual condition* of the patient, while the latter represents the *knowledge* the medical staff has about the patient. The knowledge may be outdated as the values in the patient may have changed.

The plan states known by Asbru are defined in the specification ‘plan-state-basic’, which is enriched by ‘plan-state’ to included additional concepts to summarise some of the plan states, e.g., ‘terminated’ summarises the states ‘completed’, ‘rejected’, and ‘aborted’. The synchronisation between plans is specified in ‘plan-com’, which gathers the signals that may be sent from a super-plan to its respective sub-plans. The signals are represented in internal variables to shield them from the environment which simplifies the sequents and their proofs as environmental non-interference does not have to be specified separately.

The interface to the Asbru specification is an algebraic type ‘asbru-def’ in KIV, which simply defines a structure of the form in Figure 14. Each Asbru plan is transformed into KIV using the algebraic function ‘mk-asbru-def’ by filling in the values used by the Asbru plan for its parameters.

```
asbru-def = mk-asbru-def
  (. .filter : asbru-condition;
   . .setup : asbru-condition;
   . .suspend : asbru-condition;
   . .reactivate : asbru-condition;
   . .complete : asbru-condition;
   . .abort : asbru-condition;
   . .type : plan-type;
   . .retry : bool;
   . .subplans : string-list;
   . .wait-for : wait-for;
   . .opt-wf : bool;
  );
```

Figure 14. Syntax of Asbru plans using ‘mk-asbru-def’.

Appendix B - Symbolic execution of Asbru

This appendix gives a bit more details about reasoning about Asbru plans in KIV. More details about the symbolic execution is described in the Protocure deliverable [35] and technical report [36].

The proof method in KIV is based on a sequent calculus with rules of the form:

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\, name.$$

Rules are applied bottom-up. Rule *name* refines a given conclusion $\Gamma \vdash \Delta$ with n premisses $\Gamma_i \vdash \Delta_i$. Furthermore, KIV uses rewrite rules to rewrite sub-formulas, which are of the form

$$name: \quad \phi \leftrightarrow \psi,$$

to replace a formula ϕ by an equivalent formula ϕ anywhere within a given sequent.

The idea of symbolic execution of arbitrary temporal formulas (e.g., Asbru plans) is to normalise the temporal formulas to the form $\tau \wedge \circ\, \phi$, which separates the possible first transitions from the temporal formulas describing the system in the next state. The general pattern of the normal form is given by

$$\tau_0 \wedge \textbf{last} \vee \bigvee_{i=1}^{n} (\exists X_i . \tau_i \wedge \circ\, \phi_i),$$

with X_i static variables occurring both in transition τ_i and system ϕ_i to capture the link between these formulas. The operator **last** is included as the system may also terminate. The rules in KIV to rewrite arbitrary temporal formulas to normal form are described in [1].

After normalisation, the sequent can be rewritten using the rules *dis l* and *ex l* to eliminate disjunction and quantification.

$$\frac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta}\, dis\ l \qquad \frac{\phi[^{X_0}/_{X}], \Gamma \vdash \Delta}{\exists X . \phi, \Gamma \vdash \Delta}\, ex\ l$$

where X_0 is a fresh static variable with respect to the variables in free(ϕ)$\backslash\{X\}\cup$free(Γ, Δ). For the remaining premises

$$\tau_0 \wedge \textbf{last} \vdash \qquad \tau_i \wedge \circ\, \phi_i \vdash$$

the two rules *lst* and *stp* can be applied

$$\frac{\tau[^{X,X,X}/_{A,A',A''}] \vdash}{\tau, \textbf{last} \vdash}\, lst \qquad \frac{\tau[^{X_1,X_2,A}/_{A,A',A''}], \phi}{\tau, \circ\, \phi \vdash}\, stp$$

where X, X_1, X_2 are fresh with respect to free(τ, ϕ). Note that rule *lst* deals with the situation when execution terminates and all free dynamic variables A - no matter if they are unprimed, primed, or double primed - are replaced by fresh static variable X. The result is a formula in pure predicate logic with static variables only, which can be proven with standard first-order reasoning. The rule *stp* advances the trace one step. The values of the dynamic variables A and A' in the old state are stored in fresh static variables x_1 and X_2. Double primed variables are unprimed variables in the next state. Finally, the leading next operators are discarded and the proof method continues with the execution of ϕ_i.

Table 2. Notation

Temporal Logic Operators and Statements (Sections 5 and 6)	
$\Box\,\varphi$, $\Diamond\,\varphi$, $\circ\,\varphi$, $\bullet\,\varphi$, **last**	See Table 1
$\mathcal{B}$	Background knowledge
T	Treatment
P	Patient group
N	Medical intentions
M	Medical guideline
Drug(x)	Holds if and only if drug x is administered at that point in time
SU	Sulfonylurea drug
BG	Biguanide drug
QI	Quetelet index
$T \preceq_{\varphi} T'$	Treatment T' is at least as preferred as treatment T
Good$_{\varphi}(T, P)$, Good$_{\varphi}(M, P)$	Treatment T, respectively, medical guideline M, is in accordance with good practice medicine for patient P and criteria φ

Asbru (Sections 4 and 6.2)	
considered, possible, activated, suspended, aborted, completed	Plan states
filter, setup, complete, abort	Conditions controlling execution
consider, activate	Synchronizing signals

Specification in KIV (Sections 7 and 8)	
v_s, v_d	A static, respectively, dynamic variable, which has a constant, respectively changing, interpretation on each time point
v'_d, v''_d	v'_d is the value of v_d after a system transition, v''_d is the value of v'_d after the environment transition, i.e., the value of v_d in the next state
Knowledge(*pre*, *post*)	For patient data structures *pre* and *post*, with *pre* denoting the current state and *post* the next state of the patient, the predicate Knowledge defines the relation that must hold between *pre* and *post*
$s[v]$	The value of variable v in algebraic sequence s
$s[v, c]$	Algebraic sequence s, where v is updated with value c
AS	The internal state of the Asbru program
Tick	A macro-step in the asbru execution

Appendix C - Notation

Table 2 provides a summary of the notation used in this paper.

Perl Scripts and Monkeys: Open Source Code Quality Checking

Adriaan de Groot[*]

Abstract

The tools available for checking code quality — in the broadest sense of the word, including aspects of all of the artifacts that come out of an Open Source development project — are fragmented and are often applied in an uncoordinated fashion. This paper shows one collection of tools that is applied in a consistent and coordinated fashion to the artifacts of the KDE software project. These tools do not reach the level of sophistication of static analysis which is available in an academic setting. On the other hand, they are applied to millions of lines of source code daily and produce information that is one factor in guiding the development work for this Open Source project. Future expansions of the quality checking tools will include more sophisticated checking such as static analysis, when such tools are available under Open Source licenses and are wrapped up for use within the framework.

1 Research Context

The collection of metrics on Open Source software is increasingly popular as a research topic. Metrics related to defect density, overall quality, project quality, contributor behavior and communication are now research topics for a variety of European research projects [16, 9, 8]. Not only is the development process easy to study in Open Source projects due to the transparency of that process — most of what happens in an Open Source project happens in plain sight — but the artifacts of the process, like source code repositories, mailing list archives and bug databases are also available.

The author is involved in two research projects which examine the quality of Open Source projects, although the *purpose* of such examination differs wildly:

- The E*OS^3 project[6] aims to create a quality of service standard for Open Source software service provider. While there is a lot of literature about the topic "development processes" in general, there is very

[*] LaQuSo, University of Nijmegen. Partially supported by IST project SQO-OSS, project number 033331.

little about Open Source development processes and their impact on software quality.

The E*OS^3 standard is supposed to become a metric to measure the quality and reliability of Open Source solution providers in the SMB market, and thereby strengthening their positions with regards to bigger service partners in the market.

The E*OS^3 project looks at different aspects of the quality of a delivered service, with en emphasis on processes rather than projects. After all, most projects involving Open Source products really are about integrating Open Source components into existing software stacks.

- The SQO-OSS [16] project aims at finding and implementing quality metrics that can be used (for instance within EOS^3) in quality evaluation: using the public data available on projects to find out about the quality of an Open Source codebase.

  Data that can be used for that are for example mailinglist archives to extract sociological data, the sourcecode itself to extract data such as number of lines of code, programming languages used — generally all kinds of data that can be gathered by analysing the data available.

  Data- and text mining will be used to extract all kinds of quality-related information. This information will be analysed and used to improve the quality of the product to close the feedback loop. The long-term goal of the SQO-OSS project is creating tools that feed the data gathered back into the development cycle. One can think of a plugin for a development environment that provides near-realtime information about the impact on quality a commit has.

The broad range of existing research projects shows that the quality of Open Source projects is a popular topic today. The projects financed by the European Union intend to produce Open Source tools and platforms to do the evaluation; measuring quality using Open Source tools will be all the easier when these projects have run their course.

2 Development Context

Quality checking is an essential part of software development. No software engineering text would be complete without a chapter on testing, unit testing, norms, procedures, source control and quality assurance. The CMM [2] and OSMM [11] software development maturity models include explicit examinations of the quality checking part of a given development project.

Oddly enough, quality assurance and coding guidelines are not always part of Open Source projects. Establishing such procedures and norms is easily forgotten in the early stages of such projects, and imposing them afterwards is quite difficult. Naturally there are projects such as Mozilla and Apache which have stringent norms; these are the exception rather than the norm.

In a large project (some millions of lines of source code) the imposition of coding styles *a posteriori* is further confounded by the sheer size of the task. Just *finding* all of the problems is a vast task. For this reason, tools are needed. Such tools should spot problems in the source code and other artifacts of the project that indicate quality problems. Those problems may be real bugs, maintainence hazards, or sub-optimal code.

The research metrics on software (project) quality described in the previous section are of a fairly high level of abstraction. The aggregate numbers do not report *specific* defects to the developers of a project. If the quality of a given project is negatively affected by some particular construction (bad code, for instance) then the most valuable thing that a metric-calculating tool can do for the project is report which construction it is. The more specific the report is, the more easily developer effort can be focused on repairing the defects. This suggests that the metrics tools should — in order to improve overall quality — report results in an expansive manner to the developers. The SQO-OSS project intends to do so.

In the KDE project [10], which has existed since 1996 without any strict coding style or explicit quality assurance, the introduction of automated quality checking tools with little sophistication (hence the “monkeys” in the title of this paper) has improved the quality assurance situation somewhat. The improvement comes from various sources:

- The use of buildbots (systems that continuously build the software), dashboards (display systems for any errors from the buildbot) and build farms (large coordinated collections of servers for compilation) mean that the source code is exercised far more in different configurations and the results fed back to the developers

- Increasing interest in unit tests. Unfortunately many of the unit tests must be created manually, which is a daunting task. Additionally, many parts of KDE are not the kind of programs that are easily unit-testable: interactive graphical applications do not support much testing theory (as opposed to, say, ADTs).

- The use of code checking tools to spot poor code or poor code constructions. The next section of this paper describes the system that the KDE project uses for checking for bad code.

The use of any set of tools within a large distributed project depends on the availability of the tools to everyone *or* the availability of the results of those tools to everyone. For CPU intensive checks it is most convenient to present the results on a website somewhere.

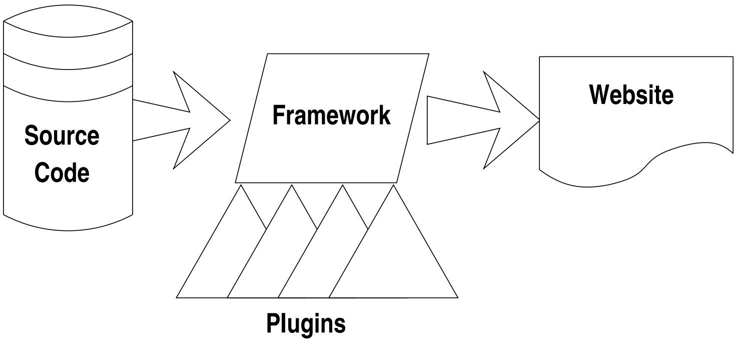


Figure 1: A high-level view of the architecture of the EBN system. A source checkout of the source code to be examined is examined by the top-level scripts which then apply small scripts to the source code.

3 English Breakfast Network

The English Breakfast Network [18] is the popular name given to the KDE [10] source code quality checking framework. This system is the prototype implementation of a code quality checking framework. The system does the following several times a day (as far as CPU use on the system it runs on permits):

- Check the user documentation
- Check the API documentation
- Check the source code

Each of these parts of the English Breakfast Network system (EBN) checks one part of the development artifacts against a collection of guidelines specific to that part. The checks are described in the following sections.

The system as a whole consists of Perl scripts which coordinate the actions of “monkeys,” the small plugin scripts which actually check a particular aspect. The top-level Perl scripts deal with collecting the output from the tools and making a presentation on a website out of that output. Figure 1 shows a very high-level view of the architecture of the system.

3.1 User Documentation

Within the KDE project the documentation for end users — the manuals, the reference guides — is written in XML docbook [19]. This format is one that is easily machine parsable using XML processors and there are various tool chains for converting the input docbook into pleasing output. The KDE project produces both HTML and PDF output from the XML Docbook sources. In order to do so automatically, the XML “source” must be valid at all times; this validity is checked by both the EBN tools and the conversion tools which are run regularly for production.

Besides the syntactical correctness of the text, the quality of the writing is important as well. Since this is end user documentation, it must be readable by a large audience. This places some restrictions on the language used. The XML Docbook is also used as a source for the (manual) translation of the user documentation, so the language must be clear and concise. While doing full natural language processing is out of scope of the EBN, there are some basic checks that *can* be done.

- Spell-checking. The actual language text can be extracted from the XML Docbook source by removing XML tags and entities. The resulting stream of words can be given to a spell-checker. This yields a list of misspellings.

- Grammar-checking. The guidelines for grammar are straightforward: contractions in English such as “isn’t” and “don’t” are not allowed. Looking for such errors is a straightforward pattern matching problem.

- Phrase-checking. There are a number of stock phrases used in the KDE documentation which must be adhered to. There are also fixed phrases for describing certain parts of the computer system that may be incorrect in the variety of English (British, Canadian, American, Australian) spoken by the person writing the documentation. Again, this is a pattern matching problem.

- Entity-checking. While stock phrases must be written out in the text for grammatical correctness (e.g. the phrase “double-click” may be translated with a different noun declension in different places in the document) there are also stock phrases which do not need special treatment in *any* language. These include the names of the author, names of applications,and some stock phrases that always appear as complete sentences or paragraphs (e.g. the translation of the GNU Free Documentation License). For such phrases, XML Docbook entities are defined and they should be used in the documentation itself, since that saves the translators effort.

- Translation-checking. As the documentation is intended to be translated into sixty or more languages, there are some requirements imposed by the translation framework as well. There must be a marker in the XML Docbook source for the translators; this is replaced by credits to the translators when they translate the document.

The tools used for the checking of user documentation are written in Python and C; the tool for checking preferred forms, for instance, is a 68-line glorified grep. The user documentation checks have caused some fixes to be committed to the KDE source repository, but there has been no concerted effort to reduce the number of defects reported.

3.2 API Documentation

The API documentation of a large library is of great importance to new developers who need to learn how to use the library. The basic libraries in the KDE software project are nearly one million lines of C++ code, and represent several thousand classes of public interface. The number of classes is a serious barrier to participation, which is why it is important that the API be documented in a clear and consisten manner.

In the API documentation, we stress *syntactical correctness* over the more language-oriented checks performed on the user documentation. This has a practical reason: where the user documentation is written in a highly-structured format (XML Docbook) the API documentation is free-form text that is processed by the Doxygen[4] tool. This difference in formats means that the most effective manner for processing the API documentation looking for defects is to run the Doxygen tool on the sources, producing API documentation and a log file of errors from Doxygen. These log files can then be analyzed and warning and error messages from Doxygen may be tallied.

This approach is very similar to that of a traditional “buildbot” or “dashboard” for software development. In such setups, the source code is compiled on a continual basis and errors are reported to the development team. Here, we simply use a different compiler — an API documentation compiler — instead.

Parsing and beautifying the log files from Doxygen is left to a fairly large perl script.

3.3 Code Checking

The source code of the KDE project — the actual C++ that is turned into object code and run on users’ computers — is naturally that part of the code that deserves the largest part of the attention to quality. Unfortunately, actually examining the source code for defects is something that takes considerable resources and research. Clasically, static analysis can point out a large number of problems with bugs in the code; this requires syntactic and semantic analysis of the code.

While tools such as the Stanford Checker [1] do static analysis[1], most of the static analysis tools are closed, proprietary (and often only usable on non-Free platforms as well). There do not seem to be many Open Source static checking tools that are widely deployed. As the tools are often the product of a research project, they languish when the research project ends. An illustration of this is Splint (IEEE Software, [7]) which seems to have been last updated in 2004, despite showing considerable promise as an extensible analysis tool for C.

For KDE, the biggest hurdle in analysis is the language that the project uses. C++ is not amenable to analysis; the syntax is difficult when all of the corners of the language are used, including template metaprogramming; the semantics are somewhat impenetrable, and even the available Open Source C++ compilers have only recently begun to correctly support all of the full language[2]. There do not seem to be readily available analysis tools for C++ at all. Some frameworks or supporting libraries are available, such as ELSA [5] and PUMA [12], but these do not yield a complete analysis tool.

In this light and considering the effort required to build such a tool, the EBN instead implements tools which do no semantic analysis but which do point to common inefficiencies and breaches of coding guidelines. The tool implemented in KDE is called Krazy [17]. This searches — a glorified grep again — for particular patterns that indicate coding problems. Examples of the kinds of defects that Krazy checks for are:

- Use of C-isms in C++ code. This includes the C macro symbols `TRUE` and `FALSE` where C++ has Boolean constants `true` and `false`.

- Inefficient use of datatypes. Passing large structures — typically objects of some class — by value is inefficient, and the KDE codebase has a guideline to pass by const reference. Various common operations such as adding a single character to a string can be done in multiple ways, where some of those ways are much faster than others.

These coding checks are relatively simple and are done with perl scripts that scan for defects. Other checks such as flagging the use of deprecated C-library calls[3] might easily be implemented through grep.

The coding checks are inspired primarily by the way that core KDE developers would *like* the code to look; this consensus style is slowly enforced on the codebase, and the Krazy tool flags deviations from the style. This means that some of the Krazy checks are done on the uncompiled and unpre-processed source code so that they operate on the way the code looks as it is edited. These more cosmetic checks include:

[1] The Stanford Checker itself seems to have vanished into Coverity [3]. Open Source tools for such analysis include Smatch [13] and Splint [15], but these are not widely used.

[2] The support for templates was expanded considerably between gcc 2 and gcc 4, for instance.

[3] The function `strcpy` for instance, as explained in [14].

Tool	Size	Tool	Size
147	contractions	178	copyright
167	doublequote_chars	103	emptystrcompare
88	endswithnewline	219	explicit
178	license	130	nullstrassign

Table 1: Plugins for the Krazy code checker, with sizes (in lines of perl code). The size of the plugin includes the copyright and documentation parts, so the effective size is smaller by a constant.

- Correct copyright and license headers in every file.
- Correct spelling in comments.
- Class methods may not have names starting with `slot`[4].
- Camel-casing of names.

These cosmetic checks add to the consistency of the codebase, making it easier for new developers to recognize the idioms of the code in any part of the KDE source code.

The Krazy checks are all written as Perl scripts which are driven from a central Perl script. The Krazy tools share with the user documentation sanitizer that there is a defined interface which the tools must follow; this part of the framework is therefore more extensible and flexible than the API documentation checker. Table 1 shows a selection of the Krazy tools with their program sizes.

4 Conclusion

The English Breakfast Network brings together tools which individually have little sophistication. The tools are generally Perl scripts, cobbled together by monkeys. However, the cumulative effect of these tools — their way of flagging errors and reporting them in a pleasant fashion — makes the monkeys (the coders doing the work) fix the errors more quickly. Several hundred commits in the KDE source code repository directly reference the EBN as the reason for the commit.

By providing a framework which is purely plugin based and which accepts individual UNIX executables (i.e. it uses the traditional pipeline approach of processing and re-processing text output) the EBN is easily extensible and may in future be expanded by adding “real” static analysis through the use of other Open Source tools.

[4] This is a name that reflects some implementation details of the code while adding no extra information. The methods can be renamed without the `slot` prefix.

References

[1] Stanford Checker. Extensible static analysis tools. http://metacomp.stanford.edu/.

[2] CMM. Capability maturity model for software. http://www.sei.cmu.edu/cmm/.

[3] Coverity. Static analysis of c and c++ code. http://www.coverity.com/.

[4] Doxygen. Api documentation processor for c++. http://www.doxygen.org/.

[5] Elsa. Elkhound (glr parser generator) based c++ parser. http://www.cs.berkeley.edu/~smcpeak/elkhound/.

[6] EOS^3. European open source service standard. http://www.eos3.org/.

[7] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, /2002.

[8] FLOSSMetrics. Floss projects development metrics. http://flossmetrics.org/.

[9] FLOSSmole. collaborative collection and analysis of open source project data. http://ossmole.sourceforge.net/.

[10] KDE. Project home page. http://www.kde.org/.

[11] OSMM. Open source maturity model. http://www.navicasoft.com/pages/osmm.htm.

[12] PUMA. Pure manipulator (a c++ parser). http://ivs.cs.uni-magdeburg.de/~puma/home-eng.html .

[13] Smatch!!! Linux kernel c source checker. http://smatch.sourceforge.net/.

[14] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2006.

[15] Splint. Annotation-assisted ;oghtweight static checking“. http://www.splint.org/.

[16] SQO-OSS. Software quality observatory for open source software. http://www.sqo-oss.eu/.

[17] KDE Quality Measurement System. C++ code issues. http://www.englishbreakfastnetwork.org/krazy/.

[18] KDE Quality Measurement System. Overview of statistics. http://www.englishbreakfastnetwork.org/.

[19] Norman Walsh and Leonard Muellner. Xml docbook website. http://www.docbook.org/.

Model-Driven Consistency Checking of Behavioural Specifications

Bas Graaf
Delft University of Technology
The Netherlands
b.s.graaf@tudelft.nl

Arie van Deursen
Delft University of Technology and CWI
The Netherlands
arie.vandeursen@tudelft.nl

Abstract

For the development of software intensive systems different types of behavioural specifications are used. Although such specifications should be consistent with respect to each other, this is not always the case in practice. Maintainability problems are the result. In this paper we propose a technique for assessing the consistency of two types behavioural specifications: scenarios and state machines. The technique is based on the generation of state machines from scenarios. We specify the required mapping using model transformations. The use of technologies related to the Model Driven Architecture enables easy integration with widely adopted (UML) tools. We applied our technique to assess the consistency of the behavioural specifications for the embedded software of copiers developed by Océ. Finally, we evaluate the approach and discuss its generalisability and wider applicability.

1. Introduction

System understanding is a prerequisite for modifying a software intensive system [1]. As such the (typical) absence of up-to-date design documentation hampers successful software maintenance and evolution. In this paper we address this problem for the documentation of a system's behaviour. We focus on ensuring the consistency of two types of behavioural specifications: interaction-based and state-based behavioural models. The use of such specifications is illustrated by the development process depicted in Figure 1. It is based on the well-known V-model [2] and the starting point of our research.

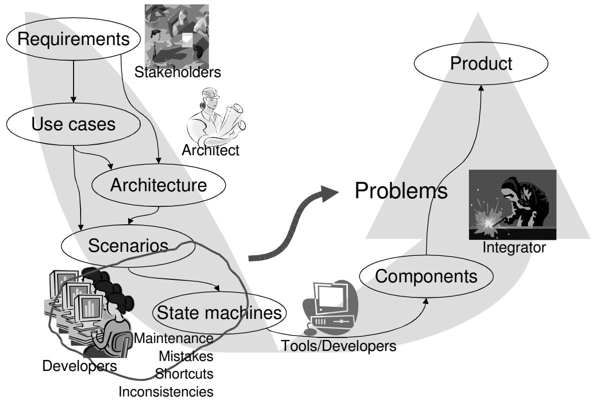


Figure 1. Typical development process

On the left branch of the 'V' analysis activities take place. Based on Requirements, the high-level Architecture is defined. This architecture identifies the main components of the system and assigns responsibilities. In parallel requirements are made more concrete by Use cases that specify typical interactions a user may have with the system. One distinctive property of use cases is that the system is considered to be a *black box* [3]. These use cases are the first interaction-based behavioural models.

Based on the use cases a set of Scenarios is defined that specifies the interactions of the system's components in terms of exchanged messages. Typically, every use case results in one (normal behaviour) or more (including exceptional behaviour) scenarios. These scenarios are also interaction-based behavioural models, but now the system is considered to be a *white-box*; they show the interactions between the components defined by the architecture.

Eventually, the architecture's components need to be implemented. This requires a complete behavioural specification. Scenarios are, however, not intended to provide such a specification for an individual component. Not only is the specification of a component's behaviour scattered across multiple scenarios, they also are usually only defined for the components' most typical and important behaviours. Therefore, a complete state-based behavioural model, a State machine, is created for each component based on the set of scenarios. This state machine is used to implement or generate the component. Finally, on the right-hand side of the 'V', the different components are integrated into a complete product.

Such a software development process, where state-based component design is based on the specification of a set of use cases, is advocated by many component-based, object-oriented, and real-time software development methods [4–7]. As such, many software development organisations deploy similar development processes.

As software evolves it is often the case that changes are made to 'downstream' software development artefacts without propagating the changes to the corresponding 'upstream' software development artefacts. This can be the result of change requests, but also of design flaws that are only discovered on a more detailed level. Even more inconsistencies are simply introduced by misinterpretations of 'upstream' development artefacts.

In this paper we focus on inconsistencies between interaction-based behavioural models and state-based behavioural models. Inconsistencies between these models can be particularly important because they decompose behaviour along different dimensions. Interaction-based models are decomposed according to the different use cases, that is, they are *requirements*-driven. State-based models, on the other hand, are decomposed according to the different components that were identified during architecture design, that is, they are *architecture*-driven. This makes it hard to discover inconsistencies [8, 9]. Furthermore, when different development groups are responsible for the development of the different architectural components, and these groups individually resolve inconsistencies in different ways, this may obviously lead to problems during integration and maintenance.

In industrial practice behavioural models are often specified as UML models. Moreover, tools are available that, based on UML, are capable of generating source code from such models. Considering such a model-based infrastructure, we believe it makes sense to view consistency checking of behavioural specifications as a model transformation problem. In this paper we investigate what the advantages and disadvantages are of using model transformation technology to discover inconsistencies between interaction-based and state-based behavioural models. Furthermore, we aim to minimise the impact of our approach on existing development processes, for instance, in terms of the languages and tools used.

In Section 2 we introduce the industrial case that motivated this paper: an embedded software control component developed by Océ, a large copier manufacturer. At Océ an important copier subsystem is developed using a process corresponding to Figure 1. Moreover, the components for this subsystem are generated from state machine models. As such, debugging, for instance, is performed on the level of state machines. As a result inconsistencies between scenarios and state machines become even more likely, making it a concern for Océ. Other work on the relation between scenarios and state machines is discussed in Section 3. The enabling technologies for our approach, as well as, the relevant part of the underlying UML specification, and our process for consistency checking are discussed in Section 4. In Section 5 we customise an existing mapping between scenarios and state machines based on Whittle and Schumann [10] for specification as model transformations and consistency checking.

Using our approach we identified several inconsistencies in the behavioural specifications of an industrial system that could lead to integration and maintenance problems. These are discussed in Section 7. Finally, we reflect on our approach in Section 8 and conclude with an overview of the contributions of this paper and opportunities for future work in Section 9.

2. Running Example

Our original motivation for investigating the consistency between interaction- and state-based behavioural models comes from a product-line architecture for embedded software in copiers developed by Océ. We use this architecture as our running example and case study, and for that reason briefly explain it first.

At Océ a reference architecture for copier *engines* is developed. In a copier both the scanning and printing subsystems are referred to as an engine. The reference architecture describes an abstract engine that can be instantiated for (potentially) any Océ copier.

As a running example we use one of the reference architecture's components: the Engine Status Manager (ESM). This component is responsible for handling status requests and status updates in the engine. ESM and the other main components of the reference architecture are depicted in Figure 2.

In a copier engine ESM communicates with two types of components: status control Clients, and Functions. Clients request engine state transitions. Requests by the external status control client (Controller) are translated by the EAI (Engine Adapter Interface) component. To perform status requests of Clients, ESM controls the status of individual Function components. Functions, in turn, recursively control the status of their composing Functions.

For the development of ESM and other engine components a process is used similar to the process outlined

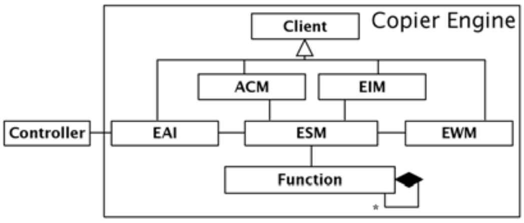


Figure 2. Architecture for copier engines

in Section 1. For this Océ relies on a model-driven approach based on UML [11]. Architects specify use case realisations using UML sequence diagrams. Based on these sequence diagrams, for every component a UML statechart diagram is created. Using special tooling[1], the source code for the engine components (e.g., ESM) is largely generated based on those statechart diagrams. For Océ's developers these statechart diagrams actually *are* the implementation.

One of the reasons for introducing a (automated) model-driven development approach was to overcome consistency problems with respect to state machine models and source code [11]. By automatically generating source code from state machines this problem is effectively moved 'upwards' to the consistency between scenarios and state machines.

For ESM, each use case addresses a specific engine state transition. A use case is accompanied by a UML sequence diagram. As an example, consider the diagram in Figure 6(a). It depicts the interaction that occurs when a copier engine is requested to go to standby, while it is running. At Océ these sequence diagrams are purely used for communication purposes, rather than input for automatic processing (e.g., model transformations, or code generation). Because of this, they are not always complete and precise. Furthermore, proprietary (non-UML) constructs are used. As an example, in these sequence diagrams the lifeline of the ESM component is decorated with the name of its (high-level) state at that point of the interaction.

To ensure successful evolution and maintenance of the reference architecture and the components it defines, a means to assess the consistency of the involved behavioural specifications is essential. It is this challenge we address in this paper.

3. Related Work

Several formal approaches have been proposed that address problems similar to ours. Lam and Padget [12] translate UML statecharts into π-calculus to determine behavioural equivalence using bisimulation. Schäfer et al. [13] presents a tool that uses model checking to verify state machines against collaboration diagrams. The use of such tools and approaches requires complete, precise and integrated interaction- and state-based behavioural models. This implies, for instance, that sending and reception of messages in scenarios are explicitly linked to events and effects in state machines. In our case, for the sequence diagrams, this is problematic. They are created early in the development process and not intended to be complete or precise.

To take this into account, we generate a state machine from a set of input scenarios, that, subsequently, is compared to the state machine that was created by the developers.

Many approaches have been defined for synthesis of state-based models from scenario-based models. Amyot and Eberlein [8], and Liang et al. [14] both evaluate over twenty of them. Evaluation criteria include languages, means to define scenario relationships and state model type. Our industrial case gives us the requirements with respect to these criteria for a synthesis approach.

Instead of using a more powerful scenario language such as live sequence charts [15], we limit ourselves to UML sequence diagrams augmented with decorations, as dictated by our industrial case study. The decorations with state information can be interpreted as conditions from which inter-scenario relationships can be derived. Finally, with respect to state model type, we consider approaches that result in state models for individual components (instead of global state models). Considering Liang et al. [14] one approach best meets these requirements [10].

Whittle and Schumann [10] present an algorithm to map UML sequence diagrams to UML statecharts. In this mapping the messages in a scenario are first annotated with pre- and postconditions on state variables, referred to as a domain theory. The mapping is based on the assumption that a message only affects a state variable if its pre- or postcondition explicitly specifies it does; the domain theory does not need to be complete. Thus, this so-called frame axiom , together with the pre- and postconditions, results in a pair of state vectors for each message (before and after). For every scenario it is checked whether it (the message ordering) is consistent with the domain theory. If not, either one can be reconsidered. Then, for each scenario a 'flat' state machine is generated for every component. Messages towards a compo-

[1] IBM Rational Rose RealTime - http://www.ibm.com/software/awdtools/developer/technical/

nent result in an event that triggers a transition; messages directed away from a component result in an action that is executed upon a transition. Loops are identified by detecting states that have unifiable state vectors. Two states vectors are unifiable if they do not specify different values for the same state variable. Subsequently, the 'flat' state machines generated for a component from different scenarios are merged by merging similar states. Two state are similar if their state vector is identical and they have at least one incoming transition with the same label. Hierarchy is added to the resulting statecharts by a user provided partitioning and (partial) ordering of the state variables.

Most work in this area focusses on the synthesis algorithm, whereas the integration in industrial practice remains implicit. In fact, many of the approaches are not supported by a tool or validated in industrial practice. Their application in practice only becomes realistic when they integrate with existing tools and standards used in industry. Therefore, we focus in this paper on UML sequence diagrams as a notation for scenarios, and UML state machines.

4. Model-Driven Consistency Checking

In this section we outline our approach for consistency checking of behavioural specifications, but, first, we introduce the technologies that enable our model-driven approach and the underlying structure of the involved behavioural models.

4.1. Enabling Technologies

Our approach takes advantage of the standards that are widely used in industry, such as UML and XMI (XML Metadata Interchange), enabling easy integration with the tools used in industrial practice. XMI provides a means to serialise UML models to be manipulated, for instance, using XSLT (Extensible Stylesheet Language Transformations). However, the XMI format is very verbose, making it a tedious and error prone task to develop such transformations [16].

OMG's Model Driven Architecture (MDA) offers, among others, a solution to this problem. MDA is OMG's incarnation of model-driven engineering (MDE). With MDE, software development largely consists of a series of model transformations mapping a source to a target model. Essential to MDE are models, their associated metamodels, and model transformations. In the case of MDA, metamodels are defined using the MetaObject Facility (MOF). The UML metamodel is only one example of such metamodels. Finally, model transformation languages are used to define transformations.

We used the Atlas Transformation Language (ATL) [17] to specify and implement the mapping between scenarios and state machines. ATL is used to develop model transformations that are executed by a transformation engine. In ATL, transformations are defined in transformation modules that consist of transformation rules and helper operations. The transformation rules match model elements in a source model and create elements in a target model. To this end the rules define constraints on metamodel elements in a syntax similar to that of the Object Constraint Language (OCL). A helper is defined in the context of a metamodel element, to which it effectively adds a feature. Helpers can be used in rules, and optionally take parameters.

The ATL transformation engine can be used with XMI serialisations of models and metamodels defined using the MOF. For the sequence diagrams and state machines in this paper we used the MOF-UML metamodel available from the OMG [18]. To create the associated models, we use a UML modelling tool supporting XMI export.

Once the source model and metamodel, target metamodel, and transformation module are defined and located, the ATL transformation engine generates the target model in its serialised form, which, in turn, can be imported in a UML modelling tool for visualisation, or serve as source model for another model transformation.

4.2. Behavioural Modelling

For the creation of interaction-based and state-based behavioural models we use UML sequence and statechart diagrams. The underlying structure of these diagrams is described by the Collaborations and State Machines subpackages of the UML metamodel. Because our transformation rules are defined on the metamodel level, we introduce them briefly. Although we discuss only simplified versions of these packages, the implementation of our technique and our case study are based on the complete UML metamodel (version 1.4 [18]).

In general the UML specification [18] allows every model element to be associated with a set of constraints. We use this to add pre- and postcondition to Messages and state invariants to states. To distinguish between preconditions, postconditions, and other constraints that might be used in the model we use stereotypes.

Source: Collaborations The Collaboration package and some other UML elements are depicted in Figure 3. In the context of a Collaboration the communication patterns performed by Objects are represented by a set of Messages that is partially ordered by the predecessor relation. For each message sender and receiver Objects are

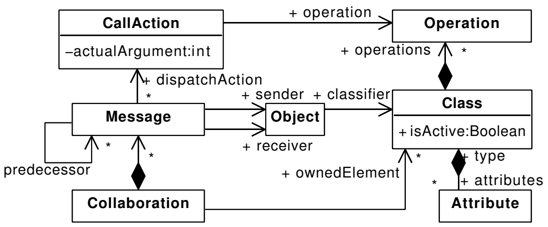


Figure 3. Collaborations (simplified)

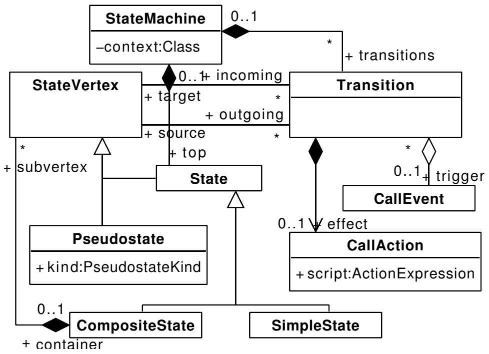


Figure 4. State machines

specified. As such, a Collaboration can be seen as the *specification* of one or more scenarios. The cause of a Message is a CallAction (dispatchAction) that is associated with an Operation. In turn, this Operation is part of the Class that is the classifier of the Object that receives the Message. Finally, a Class optionally contains Attributes that have a type.

Target: State Machines Using the (target) metamodel in Figure 4, UML state machines can be constructed that model behaviour as a traversal of a graph of state nodes interconnected by transition arcs.

A state node, or StateVertex, is the target or source of any number of Transitions and can be of different types. A State represents a situation in which some invariants (over state variables) hold. The metamodel defines the following types of States. A CompositeState contains (owns) a number of sub-states (subvertex). A SimpleState is a State without any sub-states.

Next to state nodes that describe a distinct situation, the metamodel also offers a type of StateVertex to models transient nodes: Pseudostate. Only one Pseudostate type (PseudostateKind) is relevant for the state models in this paper: initial Pseudostate. An initial Pseudostates is the default node of a CompositeState. It only has one outgoing Transition leading to the default State of a CompositeState.

Nodes in a state machine are connected by Transitions that model the transition from one State (source) to another (target). A Transition is fired by a CallEvent (trigger). The effect of a Transition specifies an CallAction to be executed upon its firing. Finally, a StateMachine is defined in the context of a Class and consists of a set of Transitions and one top State that is a CompositeState.

4.3. Consistency Checking Approach

As said, the set of scenarios is not expected to be complete or precise. For instance, when comparing, the set of scenarios and the state machines created by the developers it is unclear whether a scenario specifies universal or existential behaviour [15]. However, if we are to generate a state machine for a set of scenarios we have to take a position with respect to the meaning of those scenarios. The generation of scenarios is based on the approach in Whittle and Schumann [10]. For this, we interpret Océ's scenarios in principle as universal. This means that if the start condition of a scenario is satisfied the system behaves exactly as specified by that scenario. We consider the start condition of a scenario to be the first condition specified as decoration and occurrence of the first message. As such, the scenario in Figure 6(a) specifies exactly what happens when ESM receives the message m_SetUnit(standby) while it is in state running. However, when during execution of a scenario the start condition of another scenario is satisfied, execution continues according to that scenario. For instance, in the case of Figure 6(a), while ESM is stopping, execution could continue according to the scenario that performs the request of ESM going back to running while it was stopping.

In our approach we use model transformations for the generation of a state machine from a set of scenarios. The specification of those transformations is discussed in Section 5. To include all required information, the source model has to comply to a set of modelling conventions. When considering an arbitrary industrial case (e.g., Océ's reference architecture), the models used typically do not comply to those conventions. Therefore, we first require models to be normalised. This is discussed in Section 6.

Finally, the generated state machine is compared to the state machine that was already developed based on the same set of scenarios, the implementation state machine. Because the sequence diagrams are created early on in the development process, it is not expected that they are exactly covered by the state machines. Therefore, mismatches are expected between the generated and implementation state machine with respect to transition labels and order. This makes automating the comparison step particularly difficult. For now we manually compare the generated and implementation state machine and mainly focus on inconsistencies with respect to top-level states and transitions.

As such, we use three steps to check to consistency of behavioural specifications: normalise, transform, and compare. In the current approach only the transformation is automatic. Furthermore, the normalisation step is context-specific as it depends on the type of input models.

5. Generating State Machines

Given the source and target metamodels discussed in the previous section, we now describe how to instantiate source models, as well as the mapping between source and target models, expressed as ATL model transformations. We published all (executable) ATL transformations that we implemented, as well as (normalised) source and target (meta)models for the ATM example of Whittle and Schumann [10] in the ATL Transformations Zoo [19].

5.1. Instantiating a Source Model

Our approach based on model transformations and UML requires that all necessary information is encoded in a UML model. Whittle and Schumann [10] requires the following information for its mapping: scenarios, a domain theory, a set of state variables, and an ordered partition of that set.

The set of scenarios is specified as sequence diagrams. The types of the interacting Objects (components) are specified in a class model. The Class that corresponds to the component of interest is marked active. All Operations involved in the relevant scenarios are also specified. The pre- and postconditions of a domain theory are applied to these Operations as stereotyped Constraints. These Constraints have the form `state variable = value`. We currently do not allow pre- and postconditions in the domain theory that refer to formal parameters, as this would require interpretation of these conditions. If necessary, such constraints can be added directly to the Messages that specify an actual parameter in the sequence diagrams.

The active Class contains an Attribute for each state variable. The partition of state variables used for introducing hierarchy is encoded by setting the visibility of all state variables included in the partition to public and the others to private. Finally, the order of the state variable Attributes on the Class represents the prioritisation of state variables (the top one having the highest priority).

5.2. Model Transformations

Our transformations generate a state machine for the component that is represented by the active Class in the source model. A scenario specifies one particular path through the state machine for that component, on which it proceeds to the next state upon each communication. We refer to the state machine that only describes that path as a 'flat' state machine.

We tailored the approach in Whittle and Schumann [10] (see Sec. 3) to account for the type of input in the Océ case, our model-driven strategy, and for our goal: consistency checking. For this reason we introduce less abstractions. This makes detecting and resolving inconsistencies more convenient. Our mapping consists of four separate steps: 1) apply domain theory, 2) generate flat state machines, 3) merge flat state machines, and 4) introduce hierarchy to merged state machine.

We formalised our mapping from scenarios to state machines as four ATL model transformations that correspond to the four steps of our mapping. Every consecutive transformation uses the target model of the previous transformation as its source model.

Together, these transformations are specified in less than 700 lines of ATL code. Before these transformations can be applied to the Océ case, a normalisation step is required, which is discussed in Section 6.

Apply Domain Theory This step is specific to our approach. Unlike Whittle and Schumann [10], but in accordance with the UML, we distinguish between pre- and postconditions on the Operations of a Class and on the CallActions associated with Messages in a sequence diagram. This has two advantages. First, it allows for simple pre- and postconditions to be specified only once (i.e., on the Operations of a Class). Second, it circumvents the need to evaluate conditions that refer to formal parameters of an Operation.

When we apply the domain theory to a set of scenarios, we simply attach the pre- and postconditions on the Operations of a Class to corresponding Messages to or from instances of that Class.

The ATL specification of this mapping is straightforward. The Constraints on an Operation are copied to Messages, via their associated CallAction. Listing 1 specifies a rule that matches all CallActions. For each it generates a CallAction, `ca_out`, in the target model and initialises its constraint feature with the constraints applied to the Operation associated with the matching CallAction. Note that the constraints are added to the constraints already applied to the matched CallAction (using the `union` operation).

The result is a set of sequence diagrams in which

```
rule ConstrainedCallAction {
  from ca_in:UML!CallAction
  to ca_out:UML!CallAction(
    operation <- ca_in.operation,
    constraint <- ca_in.operation.constraint->union(
     ca_in.constraint))
}
```

Listing 1. Applying constraints to CallActions

Constraints are applied to Messages based on the pre- and postconditions of a domain theory on Operations. See Figure 6(b) for an example.

Sequence Diagrams → Flat State Machines The next step of our approach is to generate a flat state machine for every scenario in which the component of interest plays a role. In this step we map every communication to a Transition and a target State. The source State of this transition is the target State corresponding to the previous communication of the component in the scenario. As in the approach in Whittle and Schumann [10]; if the involved communication was the receipt of a Message, we say the Transition was triggered by that Message. If the involved communication was the sending of a Message, we say the effect of the Transition was sending that Message.

Based on the pre- and postconditions applied to the Messages in the scenarios by the previous step, we calculate the state vector for each State. For this we 'propagate' pre- and postconditions through the sequence diagram by application of the frame axiom. The result is a set of flat StateMachines, in which state vectors are applied to States as a set of Constraints over state variables.

As an example, the `EffectTransition` rule in Listing 2 matches all Messages in the source model sent by the component of interest. The target pattern specifies that for each such Message (`m`) among others, a Transition (`t_effect`) and a SimpleState (`trgt`) are created in the target model. The effect and target features of the Transition element are simply initialised to the CallAction (`ca`) and SimpleState created in the same rule. The source of the Transition is initialised to the target of the Transition that correspond to the previous Message (not shown).

The constraint feature of the generated SimpleState element is initialised to the set of constraints (state invariants) that hold after the Message that matched the rule. This is determined by the `stateVector` helper. For this it applies the frame axiom (specified in the `frame` helper) subsequently to the postconditions of the current Message (`'posts'`), the preconditions of the current Message (`pres`), and the state vector after the previous Message (`stateVectorPrev`). As such conditions propagate in 'forward' direction (i.e., downwards in a sequence diagram).

```
rule EffectTransition {
  from m:UML!Message (m.sender.isActive)
  to t_effect: UML!Transition(
    effect <- ca,
    target <- trgt,
    source <- ... ),
  ae:UML!ActionExpression ( ... ),
  ca:UML!CallAction ( ... ),
  trgt:UML!SimpleState (
    name <- ae.body+'_sent',
    constraint <- m.stateVector)
}
helper context UML!Message def: stateVector : Set(UML
 !Constraint) =
  let stateVectorPrev:Set(UML!Constraint) = ... in
  let pres:Set(UML!Constraint) = ... in
  let posts:Set(UML!Constraint) = ... in
  let sv:Set(UML!Constraint) =
thisModule.frame(stateVectorPrev,thisModule.frame(
 pres,posts)) in
    if thisModule.unifiable(stateVectorPrev,pres) then
      sv
    else
      sv.debug('INCONSISTENCY DETECTED!')
    endif
;
helper def: frame(frame:Set(UML!Constraint), framed:
 Set(UML!Constraint)): Set(UML!Constraint) =
  frame->iterate(c; cs:Set(UML!Constraint)=framed |
    if cs->exists(e|e.stateVariable=c.stateVariable)
     then
      cs
    else
      cs->including(c)
    endif)
;
```

Listing 2. Message →effect Transition

Additionally the `stateVector` helper notifies the user if an inconsistency is detected between the state vector after the previous Message and the preconditions for the current Message (these sets of Constraints should be unifiable).

The `frame` helper simply iterates over the Constraints in the `frame` argument and adds every constraint involving a state variable that is not referred to in `framed` to that set.

Unlike Whittle and Schumann [10] we do not apply unification of state vectors at this stage. The declarative style of our ATL specifications results in an infinite recursion: to complete a state vector we need to know whether it can be unified with other state vectors. To determine this we have to consider state vectors in 'forward' as well as in 'backward' direction. However, the state vectors in 'forward' direction, in turn, consider state vectors in 'backward' direction because of the frame axiom strategy.

Application of this step yields a set of flat state machines for a component. As an example, consider Figure 5. It depicts the flat state machine corresponding to the sequence diagram in Figure 6(b). Note that the ex-

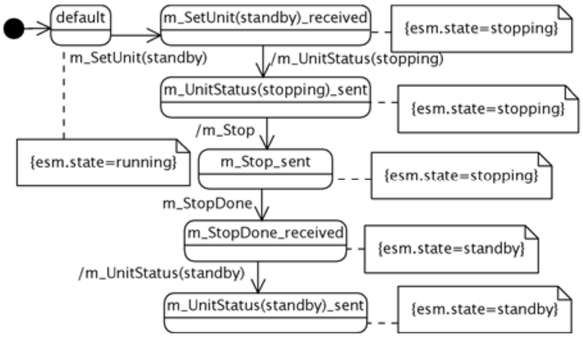


Figure 5. Flat state machine

ample only involves a single state variable and that the names of the States are derived from the particular Message that was sent or received by the component.

Merging Flat State Machines In this step we merge the flat state machines. We merge every set of states with unifiable state vectors and identical incoming transition (in terms of effect or trigger) into a single state.

Merging of states is done by the rule and helpers in Listing 3. The rule matches all states selected by the `mergedStates` helper that iteratively selects one SimpleState from every group of equal SimpleStates in the source model. A call to the `mergeable` helper results in true when 1) the receiving StateVertex and the parameter StateVertex (`s`) are unifiable, and 2) have the same name (i.e., the incoming transitions had the same trigger or effect). The `unifiable` helper evaluates to true for two sets of Constraints that do not specify different values for the same state variable, meaning that the constraint that refers to a particular state variable that is also referred to in the other set, is actually included in that set.

Transitions are matched by another rule (not shown). To discard redundant Transitions, it only matches one Transition of the Transitions between any two sets of SimpleStates that are merged.

```
rule MergedSimpleState {
  from s_in:UML!SimpleState (
    thisModule.mergedStates->includes(s_in))
  to s_out:UML!SimpleState(
    name<-s_in.name,
    constraint <- s_in.constraint)
}
helper def: mergedStates: Set(UML!StateVertex) =
  thisModule.allSimpleStates->union(thisModule.
   allPseudostates)
  ->iterate(s; mss:Set(UML!StateVertex)=Set{} |
    if mss->exists(e|(e.mergeable(s)) then
      mss
    else
      mss->including(s)
    endif)
;
helper context UML!StateVertex def: mergeable(s:UML!
 StateVertex): Boolean =
  thisModule.unifiable(self.constraint,s.constraint)
   and self.name=s.name
;
helper def: unifiable(cseq1:Sequence(UML!Constraint),
 cseq2:Sequence(UML!Constraint)): Boolean =
  cseq1->includesAll(cseq2->select(c|cseq1->collect(e|
   e.stateVariable)->includes(c.stateVariable)))
;
```

Listing 3. Merging SimpleStates

Introducing Hierarchy As suggested by Whittle and Schumann [10] we use an ordered partition of the set of state variables to add hierarchy by means of CompositeStates. The problem here, is that there is not always a matching source model element to create a CompositeState for. Therefore, we use a *called* rule (`CompositeState`). A called rule is an imperative rule that is not matched by a source model element, but is explicitly called and can have parameters. This rule creates a CompositeState for a given set of Constraints (`cseq`). These Constraints (i.e., state invariants) are determined by the `compositeStateConstraintSetsAt` helper that takes a set of Constraints that represents the current CompositeState and determines the sets of Constraints that correspond to the CompositeStates at that level. For each of those sets a CompositeState is created. This called rule is used to initialise the `subvertex` feature in the rule that matches the top CompositeState of the merged StateMachine, as well as (recursively) in the `CompositeState` rule itself. The `do` clause in the `CompositeState` rule returns the created CompositeState.

```
rule TopCompositeState {
  from cs_in:UML!CompositeState
  using {
    sm:UML!StateMachine=thisModule.allStateMachines->
     select(sm|sm.top=cs_in);
  }
  to cs_out:UML!CompositeState (
    name <- cs_in.name,
    subvertex <- sm.simpleStateStatesAt(Set{})
    ->union(sm.compositeStateConstraintSetsAt(Set{})
      ->collect(cs|thisModule.CompositeState(sm,cs))))
}
rule CompositeState (sm:UML!StateMachine, cseq:Set(
 UML!Constraint)) {
  to cs:UML!CompositeState(
    subvertex <- sm.simpleStateStatesAt(cseq)->union(
     sm.compositeStateConstraintSeqsAt(cseq)->collect(
     cs|thisModule.CompositeState(sm,cs))))
  do{cs;}
}
```

Listing 4. Adding hierarchy to state machine

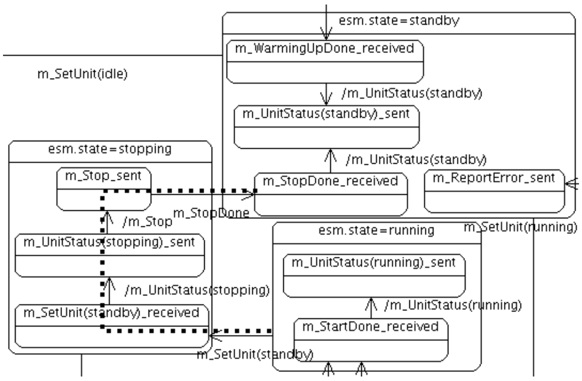


Figure 7. Merged state model of ESM (fragment)

6. Normalising the Source Model

In the case of Océ, neither a domain theory, nor a set of state variables were available. To overcome this, we normalise Océ's sequence diagrams. In particular, we interpret the decorations on object lifelines as pre- and postconditions on a single state variable: `state`. The message preceding a state decoration apparently resulted in the component moving to the indicated state. Hence, we (manually) attach a corresponding postcondition (e.g., `esm.state=starting`). A message succeeding a state decoration apparently requires the component to be in the indicated state. Hence, we attach a corresponding precondition. As an example, consider Figure 6. Finally, we added a (public) attribute, `state`, to the class corresponding to the ESM component.

7. Results

A fragment of the result of application of the transformation step to Océ's ESM component, is depicted in Figure 7. The dashed line indicates the path through the state machine that is traversed when ESM is requested to go to standby while it is running. This path corresponds to the scenario depicted in Figure 6.

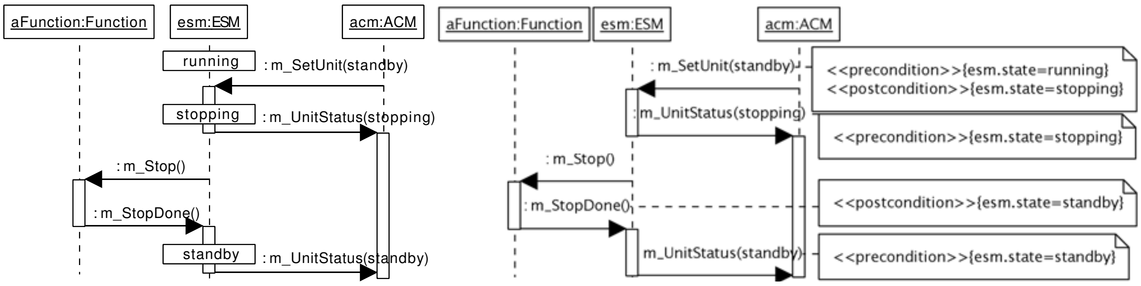
We compared this *derived* state machine with the *implementation* state machine, from which Océ generates code. There are many inconsistencies with respect to low-level states and transitions. In the implementation state machine low-level states are not only decomposed further, the sequence of states and transitions is also different in many cases. This is not surprising considering the fact that the sequence diagrams of the source model from which we derived a state machine, constitute the first behavioural model that is created for the ESM component, while, in the implementation state machine, low-level transitions and states often correspond to a single method call in the generated code. If we restrict the comparison step to the top-level states, however, the implementation state machine largely conforms to the derived state machine. Although we cannot show the implementation state machine, we were able to make several other interesting observations:

- Several transitions between top-level composite states are missing in the derived state machine. This indicates not all scenarios have been specified in a sequence diagram.
- Some top-level composite states in the derived state machine were modelled as low-level (sub) composite states in the implementation state machine. This merely indicates changes to the decomposition of states, and does not necessarily result in different behaviour.
- In the derived state machine, sometimes extra paths exists between two composite states. This indicates specific sequences of events and actions that occur in different scenarios are not specified consistently. This was the case, for instance, when two versions of a scenario existed: one for normal behaviour, and one for exceptional behaviour. For two such versions the first interactions should typically be identical (until some exception occurs), but in practice this was not the case.
- The derived state machine contains a number of unconditional transitions that form a loop, resulting in non-deterministic behaviour. This had the same cause as the previous observation.

As a response to these observations Océ could decide to add missing use cases and scenarios, and to refactor alternative sequence diagrams to remove inconsistencies in event and action sequences. Here, care must be taken, as such modifications affect the state machines of other components that play a role in the involved scenarios as well. On the other hand, if such steps are not taken and behavioural inconsistencies are only removed in the implementation state machine, other development groups, responsible for other components, might do so differently, resulting in integration and maintainability problems.

Although, the normalised source model in the Océ case only contains a single state variable, we also applied our transformation step to the ATM example in Whittle and Schumann [10][1]. This example involves three state variables. By application of our approach (in both cases) we detected several inconsistencies.

[1] Images of the (normalised) source model, as well as all (intermediate) target models for the ATM example can be downloaded from the ATL Transformations Zoo [19]



(a) Sequence diagram with decorated lifeline

(b) Normalised sequence diagram

Figure 6. Example scenario: request a copier engine to go to standby while it is running

8. Discussion

Generalisability of the approach *To a large extent our approach is generic.*

We applied our approach successfully to both Whittle and Schumann [10]'s ATM example and Océ's reference architecture. Our approach is generic with respect to input models that comply to the model conventions as outlined in Section 5.1. As such, we require a (manual) normalisation step that is context specific; it depends on the modelling conventions in use at a particular company.

Our modelling conventions are most restrictive with respect to the type of pre- and postconditions used in the domain theory. As we do not evaluate these conditions, we require them to be of the form `stateVariable=value`. In the case the conditions for an Operation refer to a formal parameter. Our approach can still be applied if the Messages associated with that Operation in the sequence diagrams specify a corresponding actual parameter. Then, we (manually) apply the condition directly to the Message in the sequence diagram and substitute the formal parameter for the actual parameter. More complicated conditions requires real interpretation of OCL expressions.

Of course, pre- and postconditions have to be available for our approach to produce more than only flat state machines. In the case of Océ's, we derived pre- and postconditions from decorations in the sequence diagrams. In general, pre- and postconditions are not always obvious from design documentation. In such situations these might have to be derived indirectly from documentation or reverse engineered from source code.

The introduction of pre- and postconditions effectively is a normalisation to the UML standard used by Océ and our tools (version 1.4 [18]). For the latest UML (version 2.0) this is not necessary, as such lifeline decorations became part of the specification (the corresponding metamodel element is called StateInvariant). To support this, only minor modifications to our ATL transformations are required.

Scalability of the approach *Our approach constitutes a first step towards fully automated consistency checking.*

In the Océ case, the source model for the transformation step includes 10 sequence diagrams that specify 62 messages. The resulting integrated, hierarchical state machine, of which a fragment was depicted in Figure 7 contains 23 transitions between 14 composite states containing in total 47 simple states.

Our approach is a first step to fully automated consistency checking of behavioural specifications. For now, we rely on manual inspection of the resulting state machine for actual evaluation of the consistency. As such, the scalability is currently not limited by the transformation steps (in the Océ case they each take less than seconds), but by the comparison step. For cases were the number of states is limited and developers have knowledge on the system, this is a feasible approach. For ESM, which is a medium-sized component (approximately 10 KLOC), this turned out not to be a problem.

Automatic consistency checking could be done by relying on naming. An example of such an approach is discussed in Van Dijk et al. [16]. It checks the consistency of the underlying XMI representations of UML models. In general this problem is equivalent to graph matching. Also for automatic approaches, however, the generation of a state machine from a set of scenarios, as discussed in this paper, is likely to be a first step.

Applicability of the approach *Our approach can be applied to iteratively develop behavioural specifications.*

We generated a state machine with the purpose of checking the consistency of different behavioural specifications. However, our approach might have other types of applications as well. A generated state machine could also be used for other types of analyses, such as model checking or performance analysis.

Next to analysis purposes, our approach is particularly also interesting for forward engineering, especially in the context of model-driven development approaches as in the case of Océ. Using our transformations based on UML, developers can easily generate different views on the behaviour of a software system or component. Furthermore, the generation not only provides insight in the consistency of the sequence diagrams with respect to each other, it also provides developers with a first candidate state machine that can be refined. As such, our technique can be applied iteratively to develop complete behavioural specifications of components: (1) specify the interactions of an initial set of use cases as scenarios, (2) generate a state machine, (3) refactor scenarios to remove inconsistencies in event and action sequences, and add missing scenarios, (4) goto step 2.

The main reason to choose for a model-driven approach based on UML for our consistency check, was the integration with Océ's development process. It circumvents the need to extract information from the MDA domain to another domain, e.g, the grammarware, or XML domain. Unfortunately, despite the availability of standards, currently available tools for (meta)modelling and transformations do not integrate well, hampering actual integration of our approach in practice. For a large part this is due to the abundance of possible combinations of XMI, UML, and MOF versions, as well as vendor specific implementation of those standards. Other problems occur due to different capabilities of modelling tools. As an example, we used Poseidon for UML to create source models because its metamodel is available from the developer's website. However, the UML models we generate do not contain layout information. Unfortunately, Poseidon is not capable of displaying UML models that do not contain layout information. As a consequence we had to use another tool for visualisation. From a large set of tools we tried, only Borland's Together is capable of generating a layout for a UML model. However, the XMI representations used by this tool are not compatible with those generated by the ATL engine. As a workaround we developed a minimal XSLT transformation that maps the XMI 'flavour' generated by the ATL engine to that of Together. An alternative is to generate the layout information required by Posedion using a model transformation.

UML vs. MOF *The use of* UML *in a limited domain makes transformation definitions unnecessary complex*

The genericity and resulting complexity of the UML metamodel result in, sometimes, inconvenient navigation through source and target models to select a certain element. Also, often relations are defined as $n : n$ while in a specific case $1 : 1$ would suffice. The result is that sets have to be converted to sequences of which the first element has to be selected. This is required very frequently, resulting in unnecessary complex ATL-code.

In cases, where only limited parts of the UML metamodel are used, an alternative could be considered. Instead of using the UML metamodel, custom MOF-based metamodels could be used, for instance, for scenarios and state machines. These metamodels could be much simpler, resulting in simpler transformation definitions.

9. Conclusions

In this paper we demonstrated the use of model transformations to check the consistency of behavioural specifications. For this we presented an approach that consist of normalisation, transformation, and comparison steps. We consider the following to be the main contributions of this paper:

- A specification of the mapping between scenarios and state machines using model transformations that is made available via the ATL Transformations Zoo [19]. An advantage of such a specification is that it can be executed by the ATL transformation engine. Furthermore, it is completely based on UML, allowing easy integration in industrial practice.
- Modelling conventions for encoding the information required for the transformation step in a single UML model. Additionally, as an example, we discussed the required normalisation step for Océ's reference architecture.
- Validation of the proposed approach by application to an industrial system, resulting in the identification of a number of inconsistencies in its behavioural specifications.

Finally, the proposed approach could be applied for other purposes than consistency checking as well, such as forward engineering and early behavioural analysis based on the generated state machine.

Currently we are extending our work with additional case studies. Furthermore, we investigate the possibilities to do consistency checking automatically. Again, by the use of MDA model transformation technologies.

Acknowledgement Part of the research described in this paper was sponsored by NWO via the Jacquard Reconstructor project. Furthermore we would like to thank Océ, and in particular Lou Somers for providing the case study.

References

[1] M. M. Lehman and L. A. Belady, eds. *Program evolution: processes of software change*. Academic Press, 1985.

[2] A.P. Bröhl and W. Dröschel. *Das V-Modell. Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Oldenbourg-Verlag, München, 2^{nd} edition, 1995.

[3] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[4] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

[5] Phillipe Kruchten. *The Rational Unified Process*. Addison-Wesley, 1998.

[6] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[7] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[8] Daniel Amyot and Armin Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1), September 2003.

[9] Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Software Engineering*, 31(12):999–1014, December 2005.

[10] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proc. 22^{nd} Int'l Conf. Software Engineering (ICSE 2000)*, pages 314–323. IEEE CS, 2000.

[11] L. A. J. Dohmen and L. J Somers. Experiences and lessons learned using UML-RT to develop embedded printer software. In *Proc. PROFES 2002*, volume 2559/2003 of *LNCS*, pages 475–484. Springer-Verlag, 2003.

[12] Vitus S.W. Lam and Julian Padget. Analyzing equivalences of uml statechart diagrams by structural congruence and open bisimulations. In *Proc. 2003 IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2003)*, pages 137–144. IEEE CS, October 2003.

[13] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking uml state machines and collaborations. In *Proc. Workshop on Software Model Checking*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 357–369. Elsevier, 2001.

[14] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proc. 5^{th} Int'l Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2006)*, pages 5–11. ACM, 2006.

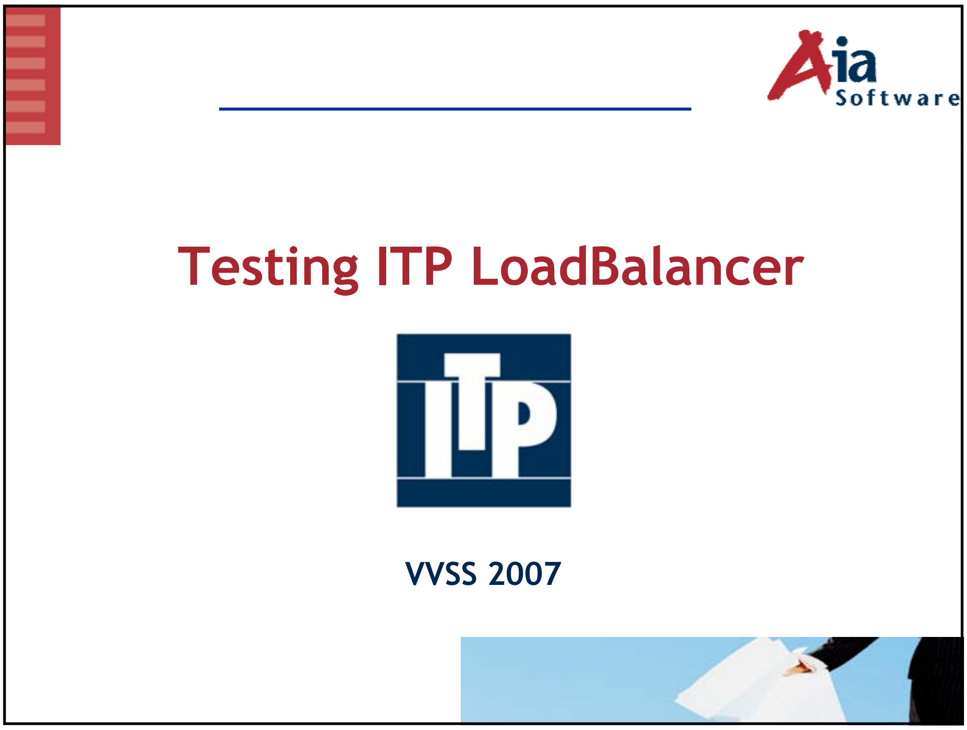
[15] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.

[16] Hylke W. van Dijk, Bas Graaf, and Rob Boerman. On the systematic conformance check of software artefacts. In *Proc. 2^{nd} European Workshop on Software Architecture (EWSA 2005)*. Springer-Verlag, June 2005.

[17] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proc. Model Transformations in Practice Workshop at MoDELS2005*, 2005.

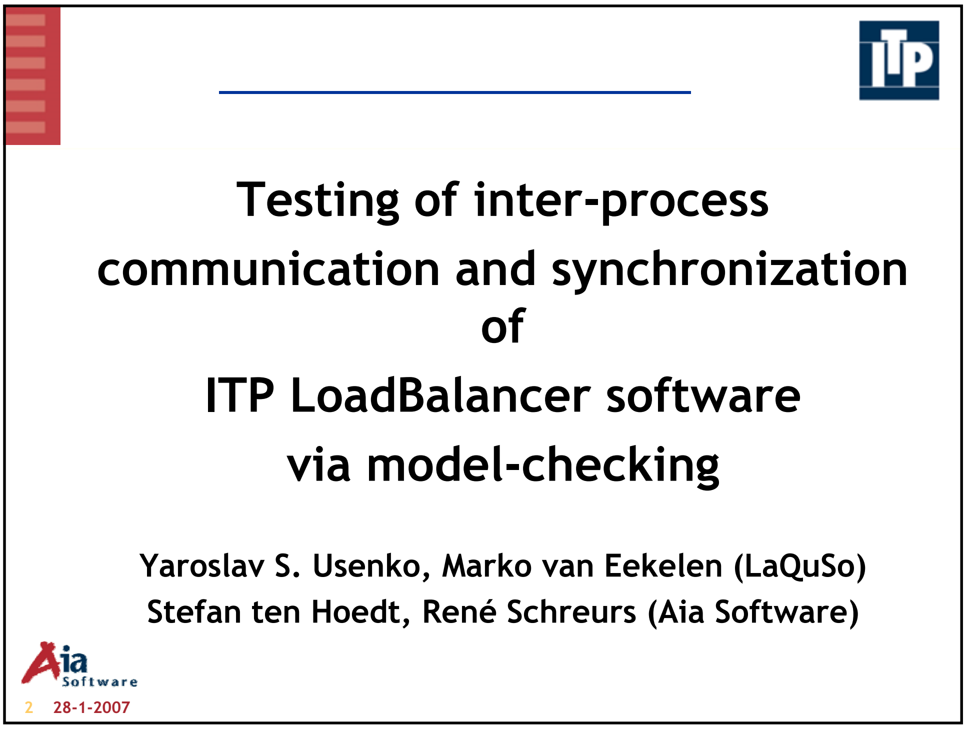
[18] OMG. OMG Unified Modeling Language Specification, Version 1.4. `http://www.uml.org`, 2001.

[19] ATL Transformations Zoo. `http://www.eclipse.org/gmt/atl/atlTransformations/#UMLSD2STMD`.



Testing ITP LoadBalancer

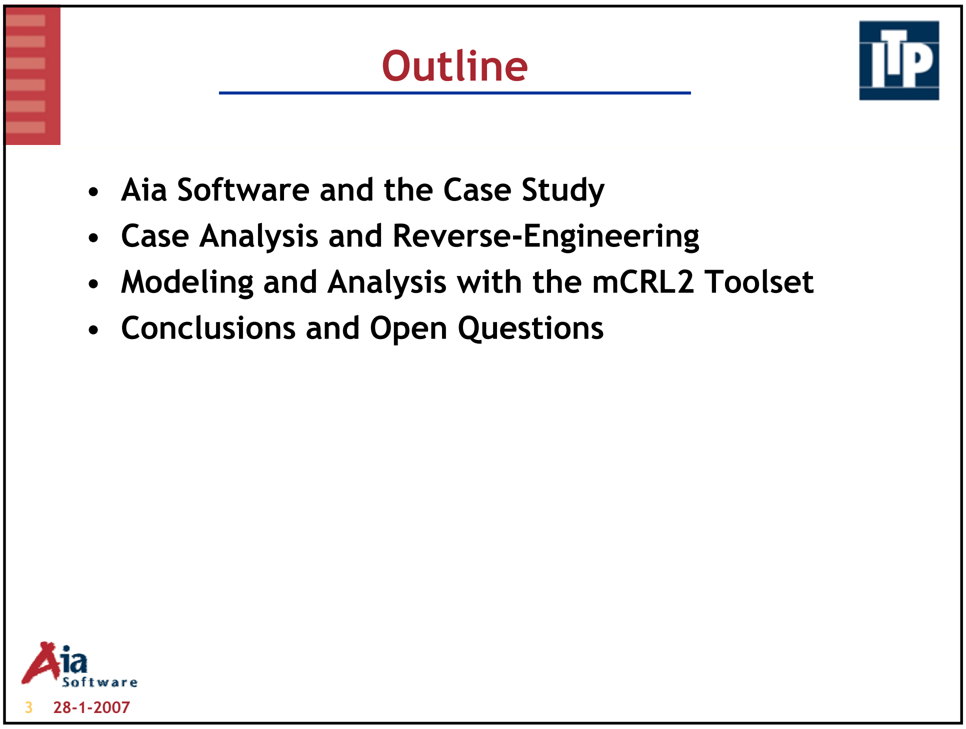
VVSS 2007



Testing of inter-process communication and synchronization of ITP LoadBalancer software via model-checking

Yaroslav S. Usenko, Marko van Eekelen (LaQuSo)
Stefan ten Hoedt, René Schreurs (Aia Software)

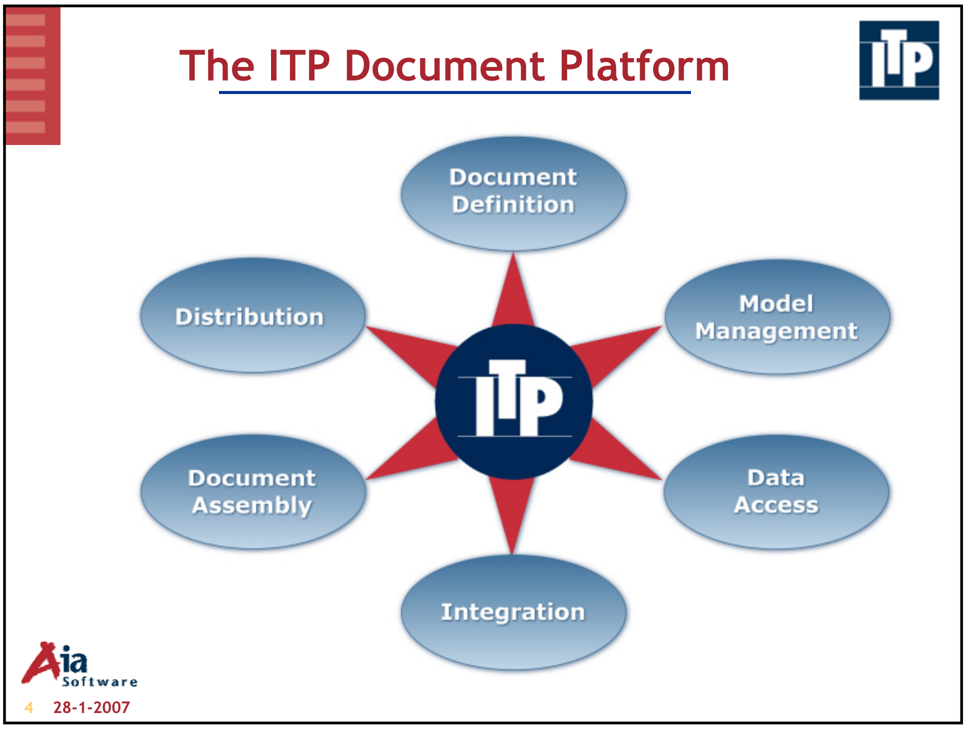
2 28-1-2007



Outline

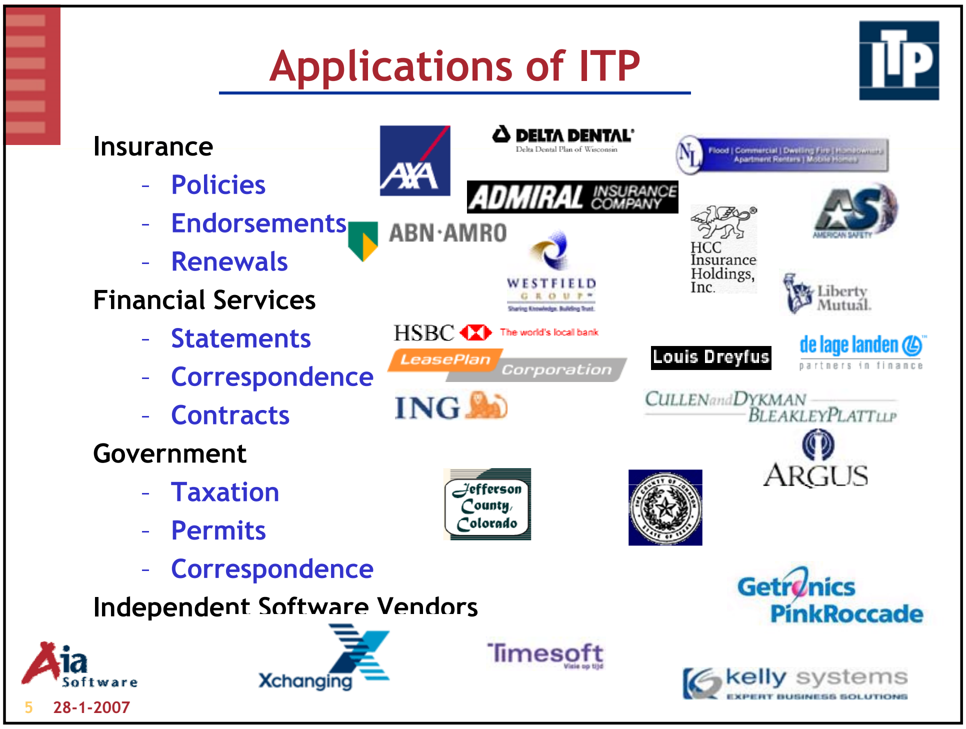
- Aia Software and the Case Study
- Case Analysis and Reverse-Engineering
- Modeling and Analysis with the mCRL2 Toolset
- Conclusions and Open Questions

3 28-1-2007



The ITP Document Platform

4 28-1-2007



Applications of ITP

Insurance

- Policies
- Endorsements
- Renewals

Financial Services

- Statements
- Correspondence
- Contracts

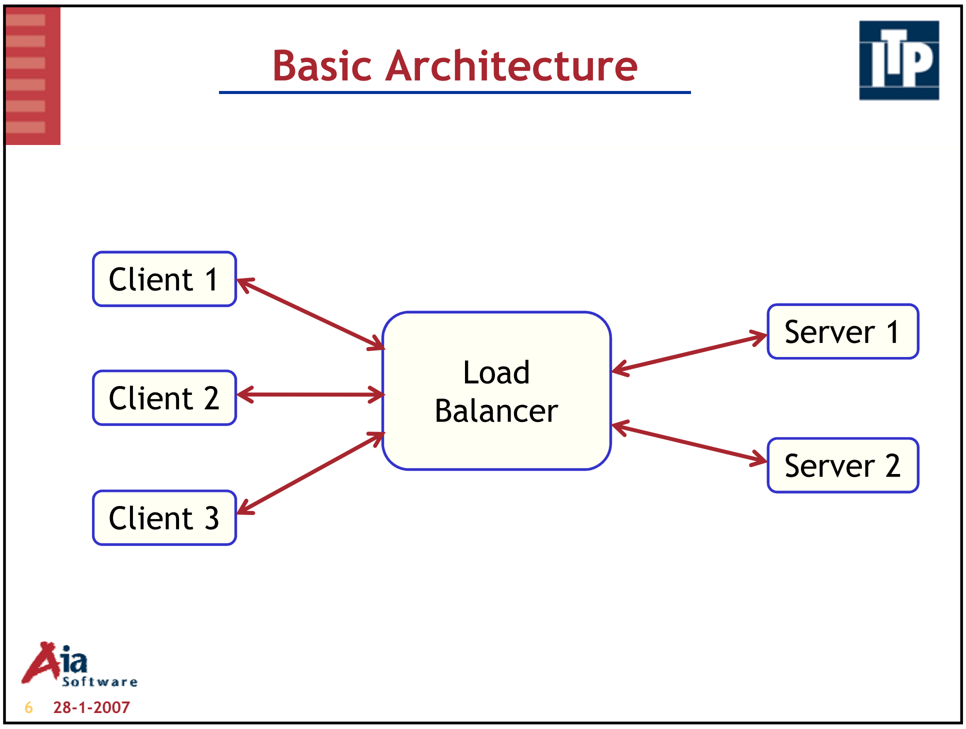
Government

- Taxation
- Permits
- Correspondence

Independent Software Vendors

ITP
AXA
DELTA DENTAL
Delta Dental Plan of Wisconsin
ADMIRAL INSURANCE COMPANY
ABN·AMRO
WESTFIELD GROUP
HCC Insurance Holdings, Inc.
AS AMERICAN SAFETY
Liberty Mutual.
HSBC The world's local bank
LeasePlan Corporation
Louis Dreyfus
de lage landen partners in finance
ING
CULLEN and DYKMAN BLEAKLEY PLATT LLP
ARGUS
Jefferson County, Colorado
Getronics PinkRoccade
Xchanging
Timesoft
kelly systems EXPERT BUSINESS SOLUTIONS
Aia Software

5 28-1-2007



Basic Architecture

Client 1

Client 2

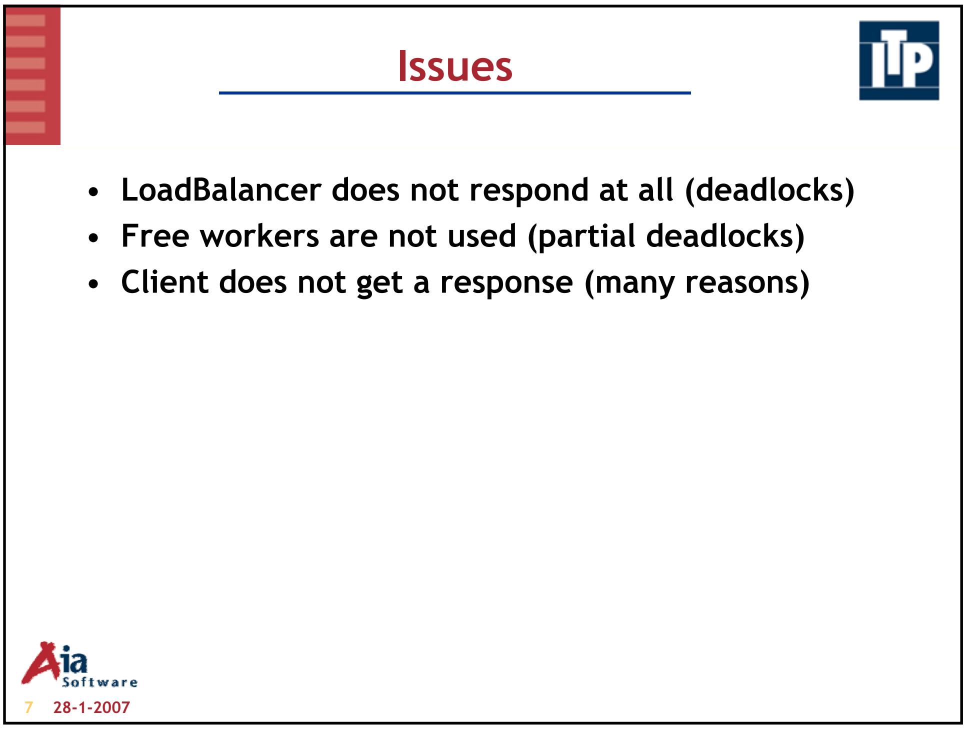
Client 3

Load Balancer

Server 1

Server 2

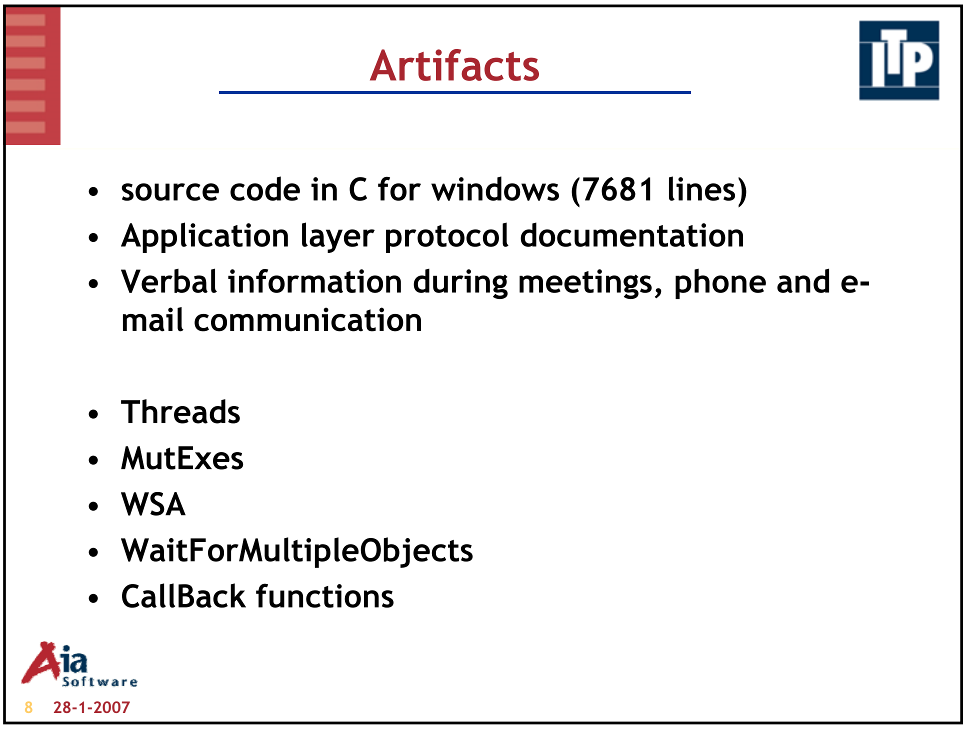
6 28-1-2007



Issues

- LoadBalancer does not respond at all (deadlocks)
- Free workers are not used (partial deadlocks)
- Client does not get a response (many reasons)

7 28-1-2007

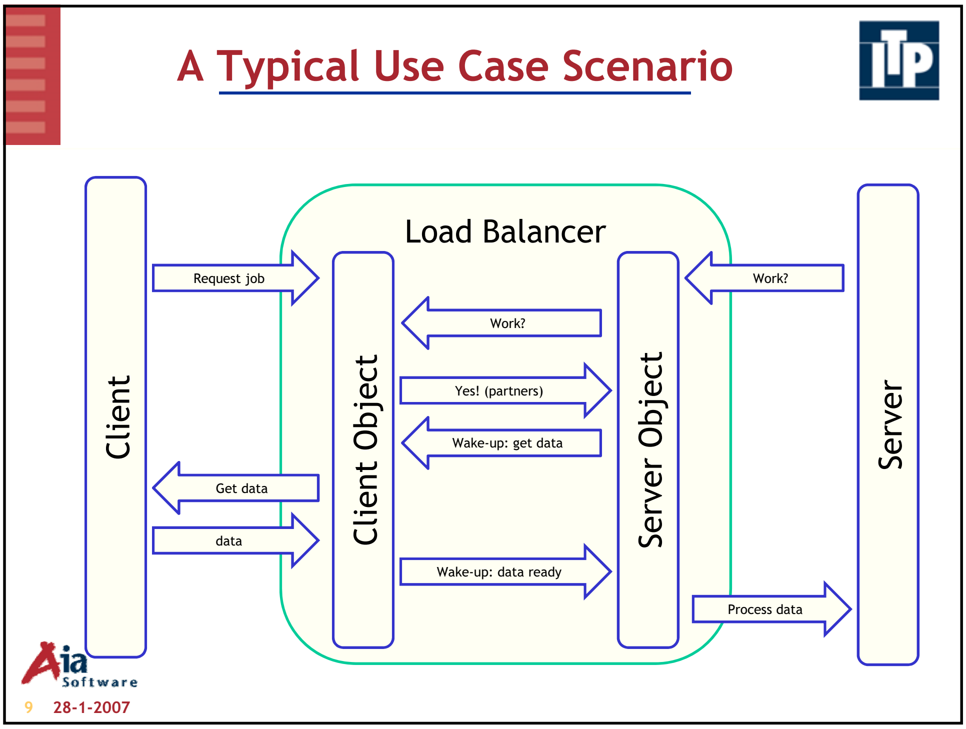


Artifacts

- source code in C for windows (7681 lines)
- Application layer protocol documentation
- Verbal information during meetings, phone and e-mail communication

- Threads
- MutExes
- WSA
- WaitForMultipleObjects
- CallBack functions

8 28-1-2007



A Typical Use Case Scenario

Load Balancer

Client

Client Object

Server Object

Server

Request job

Work?

Work?

Yes! (partners)

Wake-up: get data

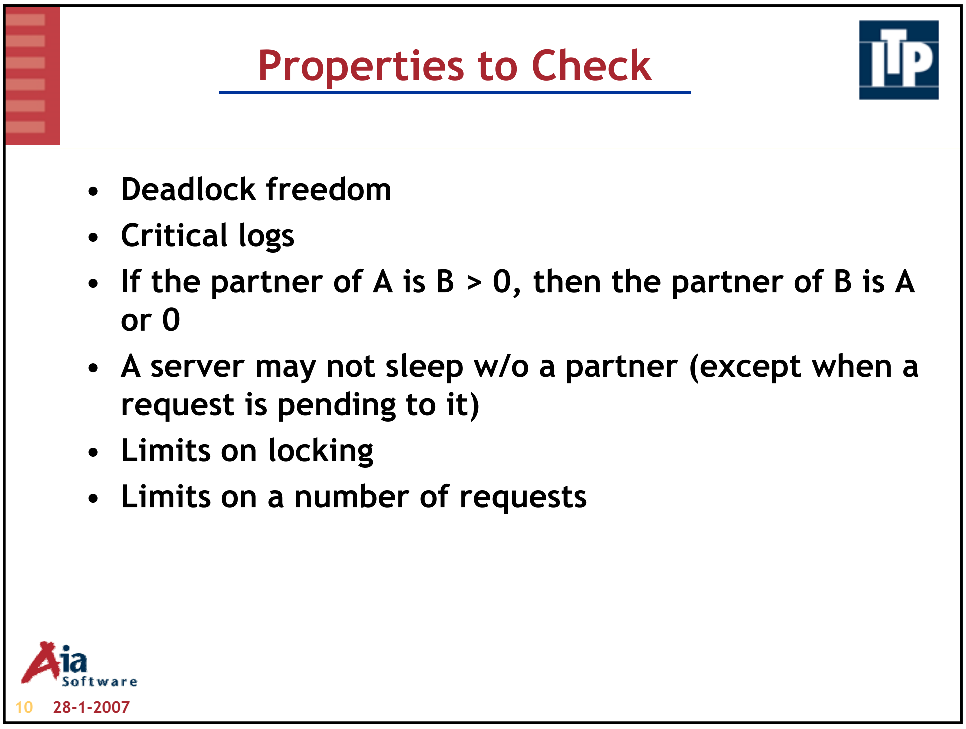
Get data

data

Wake-up: data ready

Process data

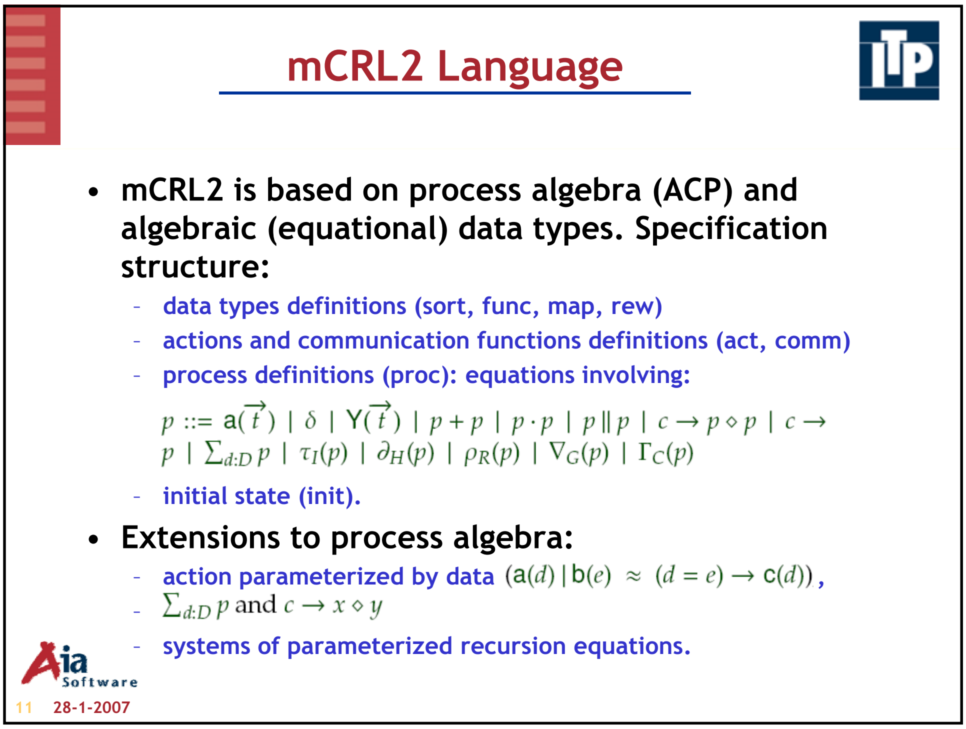
9 28-1-2007



Properties to Check

- Deadlock freedom
- Critical logs
- If the partner of A is B > 0, then the partner of B is A or 0
- A server may not sleep w/o a partner (except when a request is pending to it)
- Limits on locking
- Limits on a number of requests

10 28-1-2007

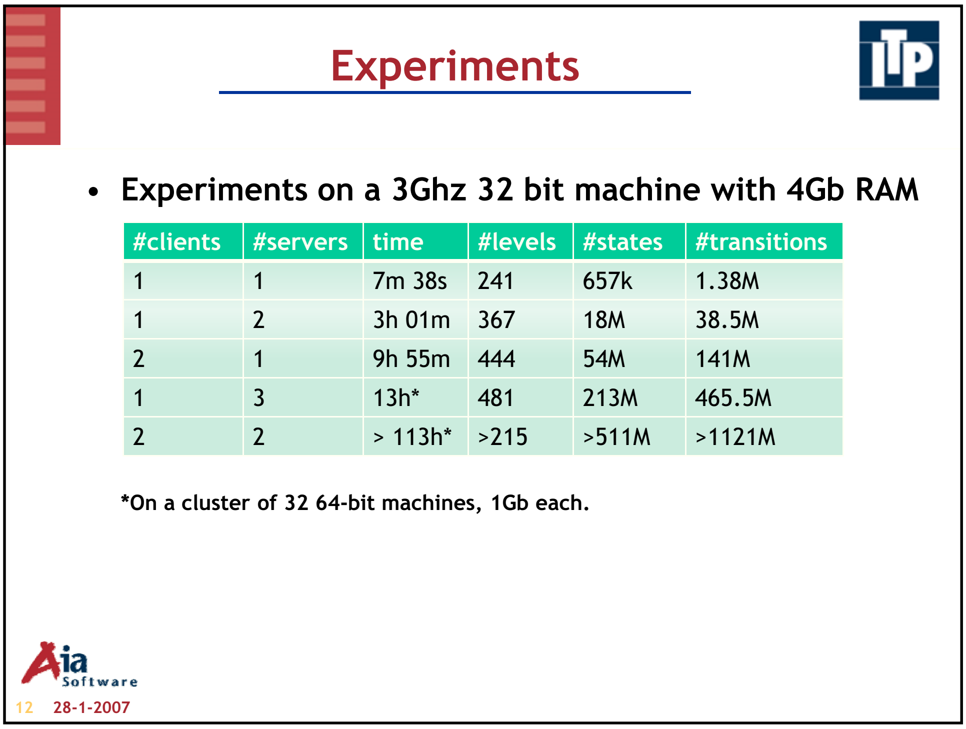


mCRL2 Language

- mCRL2 is based on process algebra (ACP) and algebraic (equational) data types. Specification structure:
  - data types definitions (sort, func, map, rew)
  - actions and communication functions definitions (act, comm)
  - process definitions (proc): equations involving:

    $p ::= \mathsf{a}(\vec{t}) \mid \delta \mid \mathsf{Y}(\vec{t}) \mid p+p \mid p\cdot p \mid p \| p \mid c \to p \diamond p \mid c \to p \mid \sum_{d:D} p \mid \tau_I(p) \mid \partial_H(p) \mid \rho_R(p) \mid \nabla_G(p) \mid \Gamma_C(p)$
  - initial state (init).
- Extensions to process algebra:
  - action parameterized by data $(\mathsf{a}(d) \,|\, \mathsf{b}(e) \;\approx\; (d=e) \to \mathsf{c}(d))$,
  - $\sum_{d:D} p$ and $c \to x \diamond y$
  - systems of parameterized recursion equations.

11 28-1-2007



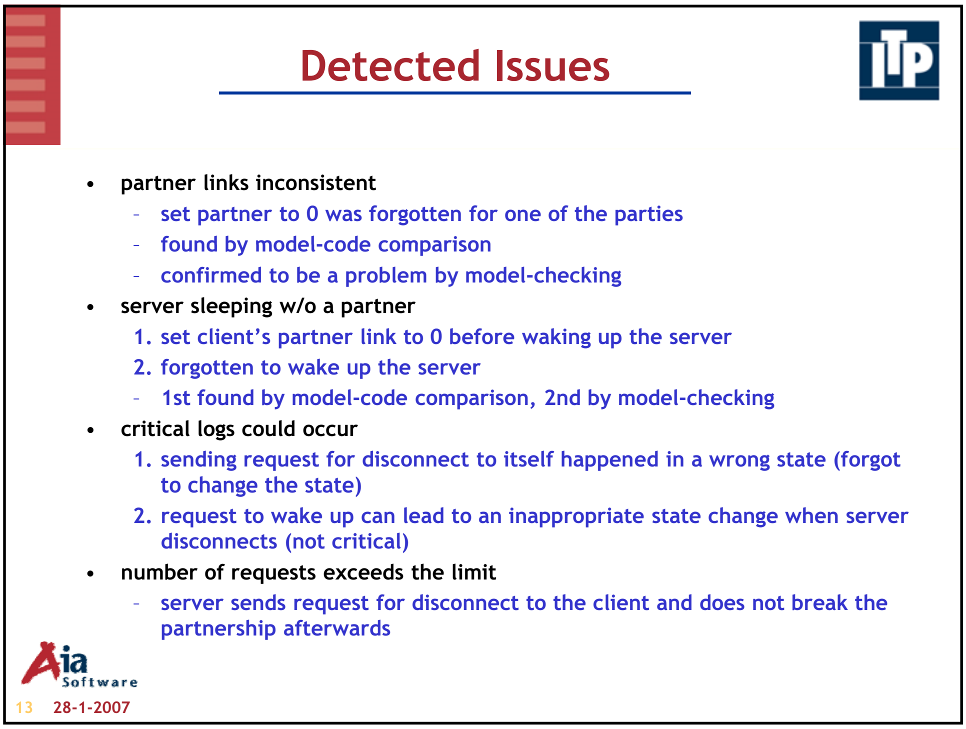
Experiments

- Experiments on a 3Ghz 32 bit machine with 4Gb RAM

#clients	#servers	time	#levels	#states	#transitions
1	1	7m 38s	241	657k	1.38M
1	2	3h 01m	367	18M	38.5M
2	1	9h 55m	444	54M	141M
1	3	13h*	481	213M	465.5M
2	2	> 113h*	>215	>511M	>1121M

*On a cluster of 32 64-bit machines, 1Gb each.

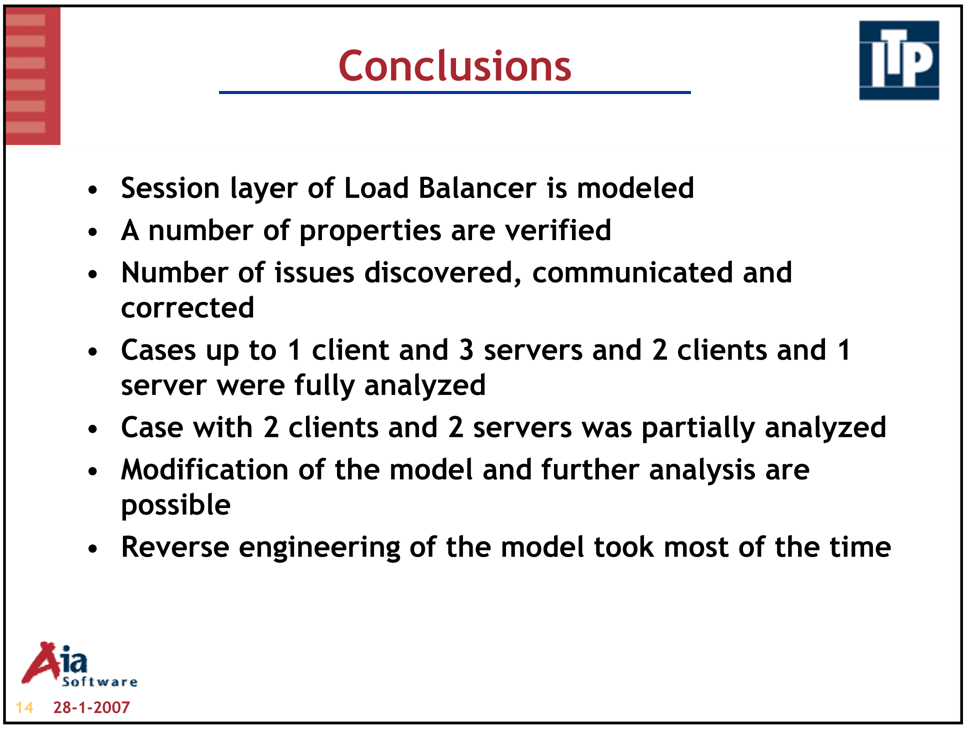
12 28-1-2007



Detected Issues

- partner links inconsistent
  - set partner to 0 was forgotten for one of the parties
  - found by model-code comparison
  - confirmed to be a problem by model-checking
- server sleeping w/o a partner
  1. set client’s partner link to 0 before waking up the server
  2. forgotten to wake up the server
  - 1st found by model-code comparison, 2nd by model-checking
- critical logs could occur
  1. sending request for disconnect to itself happened in a wrong state (forgot to change the state)
  2. request to wake up can lead to an inappropriate state change when server disconnects (not critical)
- number of requests exceeds the limit
  - server sends request for disconnect to the client and does not break the partnership afterwards

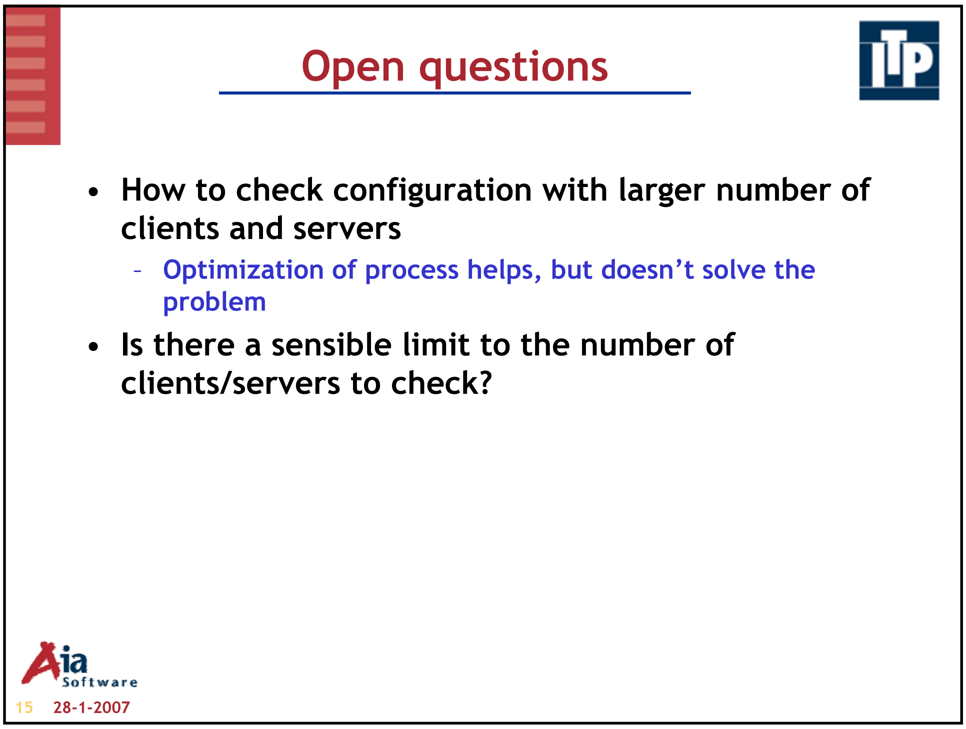
13 28-1-2007



Conclusions

- Session layer of Load Balancer is modeled
- A number of properties are verified
- Number of issues discovered, communicated and corrected
- Cases up to 1 client and 3 servers and 2 clients and 1 server were fully analyzed
- Case with 2 clients and 2 servers was partially analyzed
- Modification of the model and further analysis are possible
- Reverse engineering of the model took most of the time

14 28-1-2007



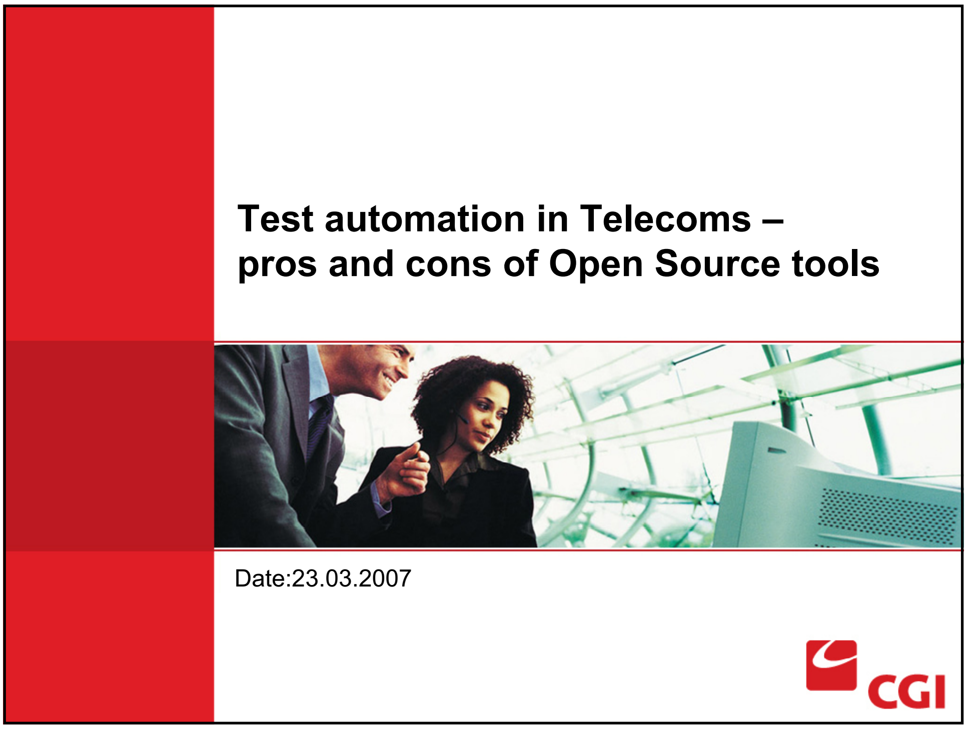
Open questions

- How to check configuration with larger number of clients and servers
  - Optimization of process helps, but doesn't solve the problem
- Is there a sensible limit to the number of clients/servers to check?

15 28-1-2007

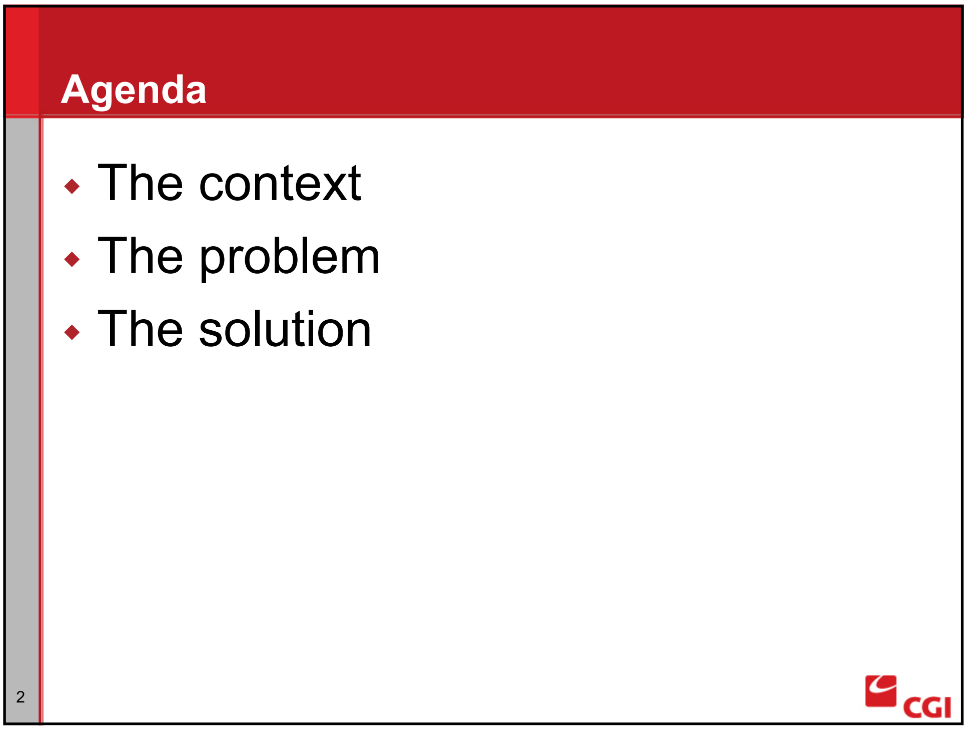
WWW.AIA-ITP.COM

INTELLIGENT TEXT PROCESSING



Test automation in Telecoms – pros and cons of Open Source tools

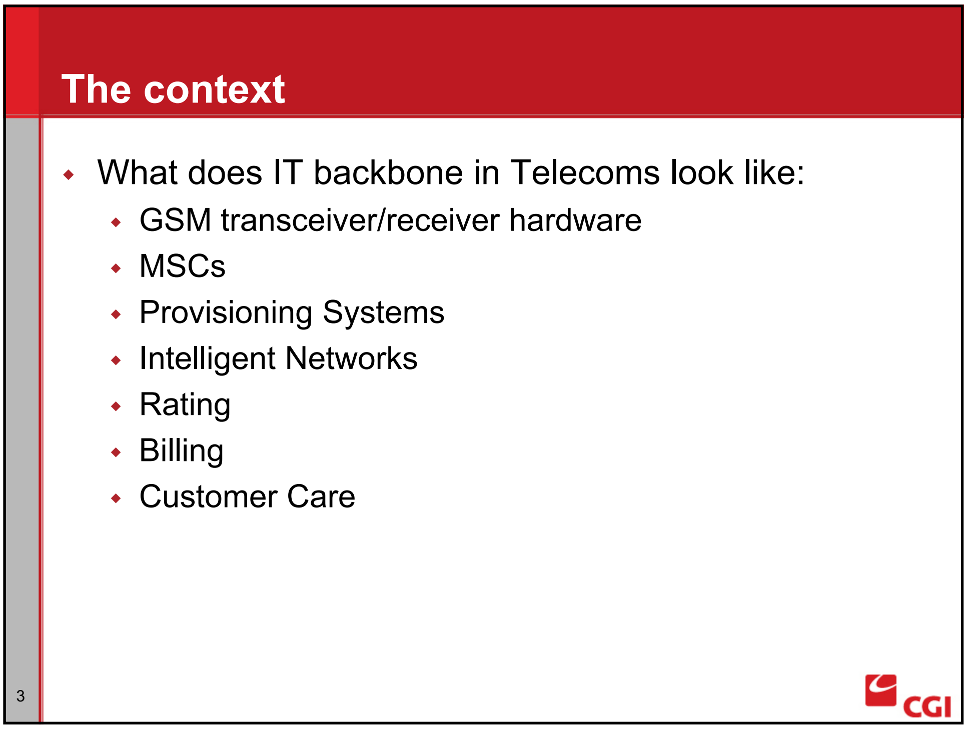
Date:23.03.2007



Agenda

- The context
- The problem
- The solution

2

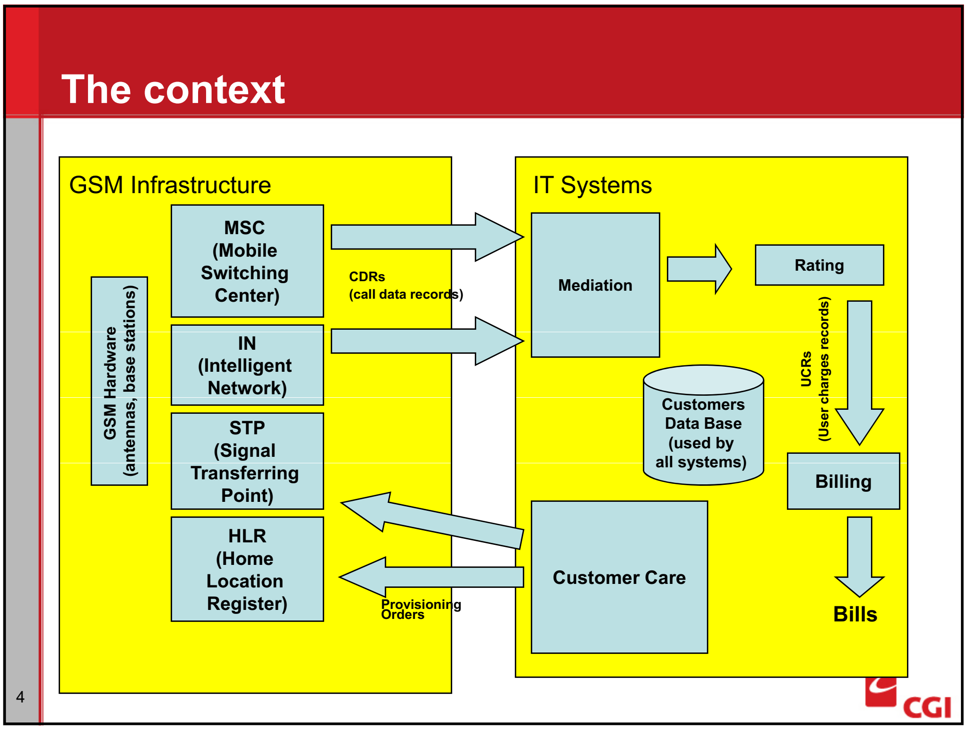


The context

- What does IT backbone in Telecoms look like:
  - GSM transceiver/receiver hardware
  - MSCs
  - Provisioning Systems
  - Intelligent Networks
  - Rating
  - Billing
  - Customer Care

3

CGI



The context

GSM Infrastructure

GSM Hardware (antennas, base stations)

MSC (Mobile Switching Center)

IN (Intelligent Network)

STP (Signal Transferring Point)

HLR (Home Location Register)

CDRs (call data records)

Provisioning Orders

IT Systems

Mediation

Rating

UCRs (User charges records)

Customers Data Base (used by all systems)

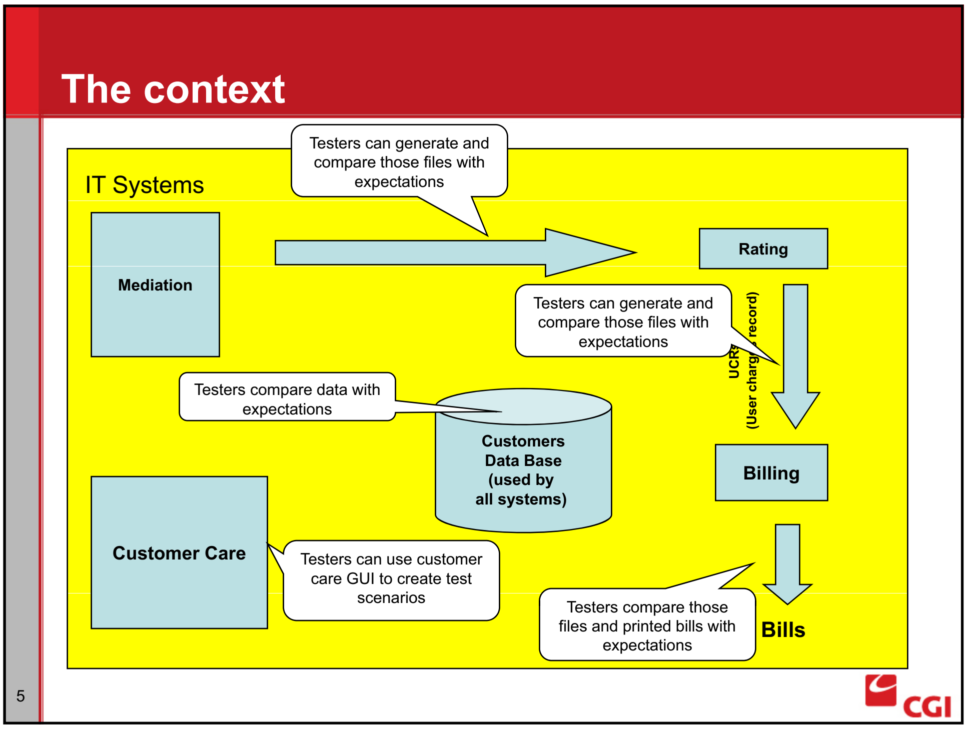
Billing

Customer Care

Bills

4

CGI



The context

Testers can generate and compare those files with expectations

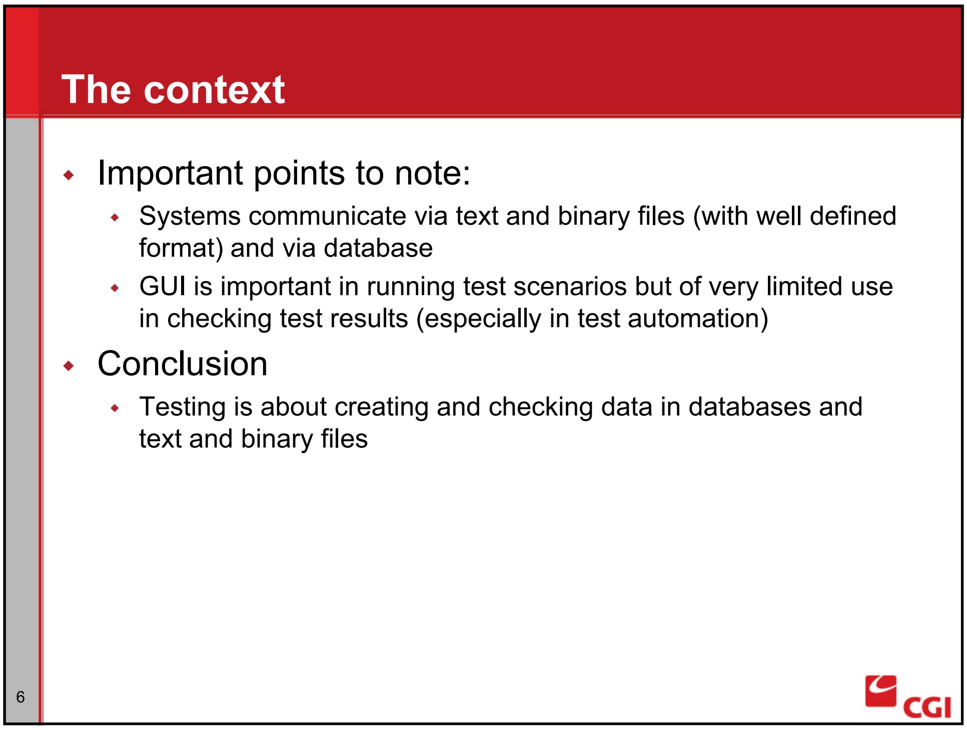
Testers can generate and compare those files with expectations

Testers compare data with expectations

Testers can use customer care GUI to create test scenarios

Testers compare those files and printed bills with expectations

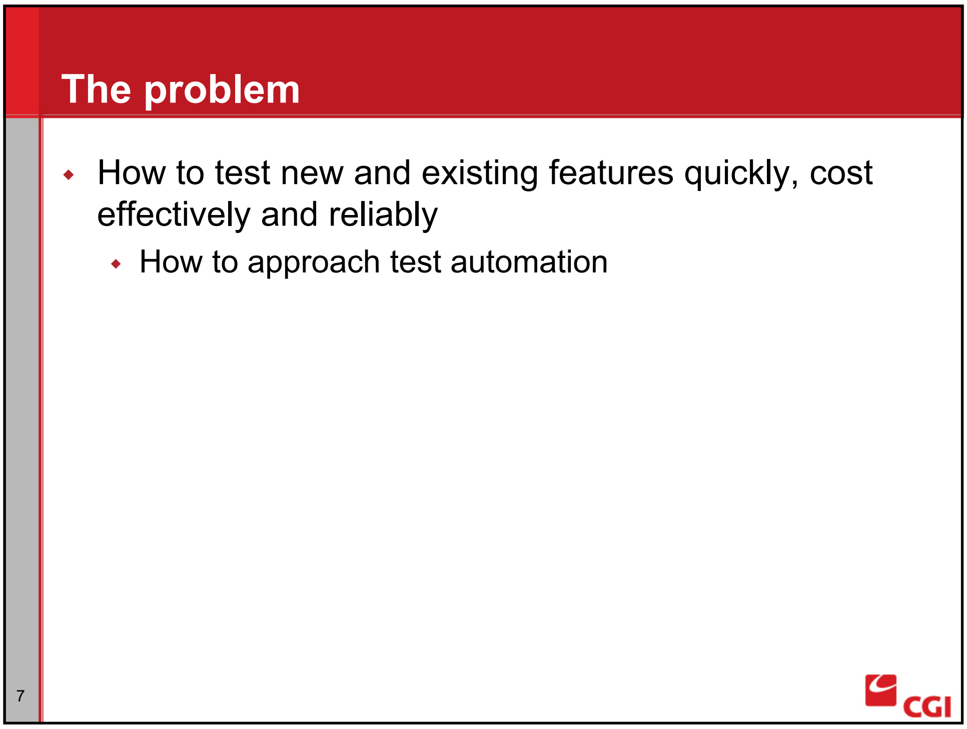
5



The context

- Important points to note:
  - Systems communicate via text and binary files (with well defined format) and via database
  - GUI is important in running test scenarios but of very limited use in checking test results (especially in test automation)
- Conclusion
  - Testing is about creating and checking data in databases and text and binary files

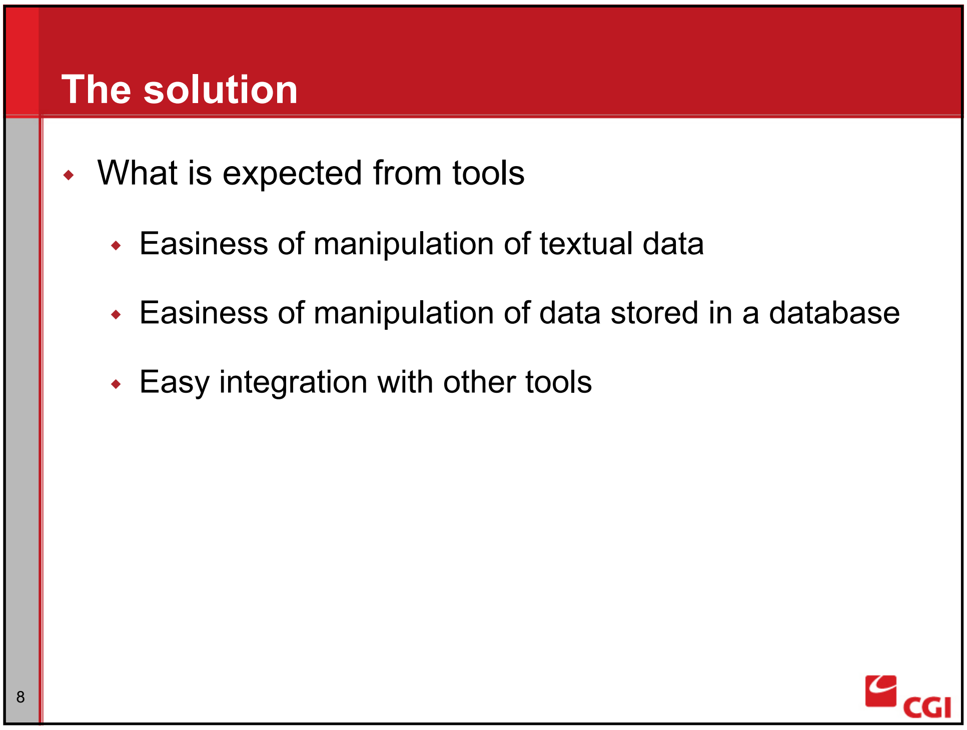
6



The problem

- How to test new and existing features quickly, cost effectively and reliably
  - How to approach test automation

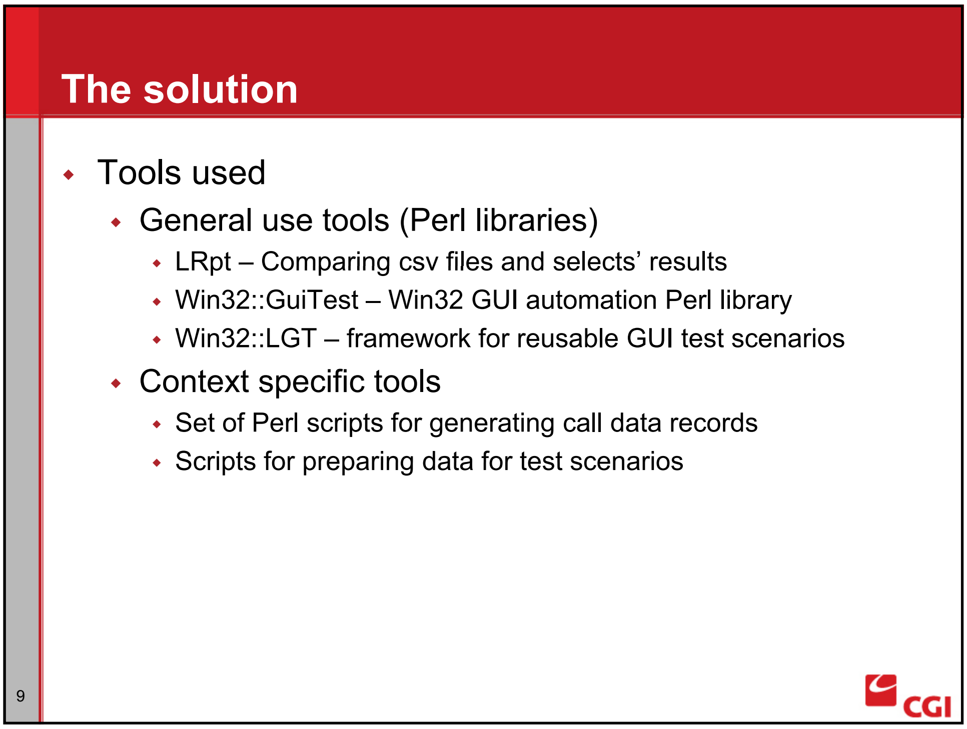
7



The solution

- What is expected from tools
  - Easiness of manipulation of textual data
  - Easiness of manipulation of data stored in a database
  - Easy integration with other tools

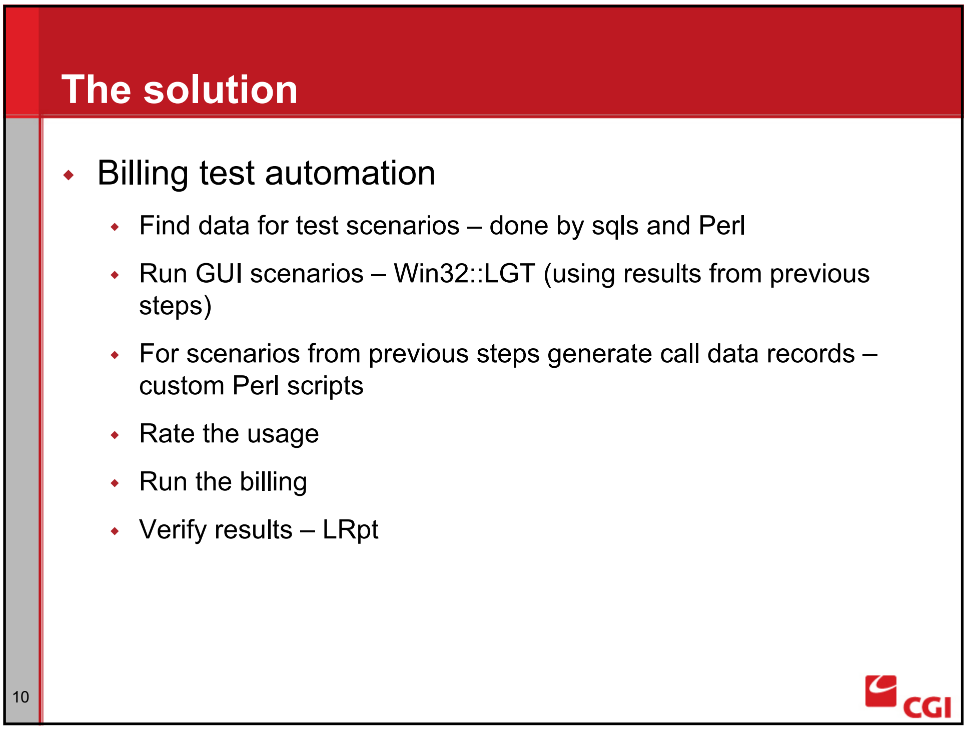
8



The solution

- Tools used
  - General use tools (Perl libraries)
    - LRpt – Comparing csv files and selects’ results
    - Win32::GuiTest – Win32 GUI automation Perl library
    - Win32::LGT – framework for reusable GUI test scenarios
  - Context specific tools
    - Set of Perl scripts for generating call data records
    - Scripts for preparing data for test scenarios

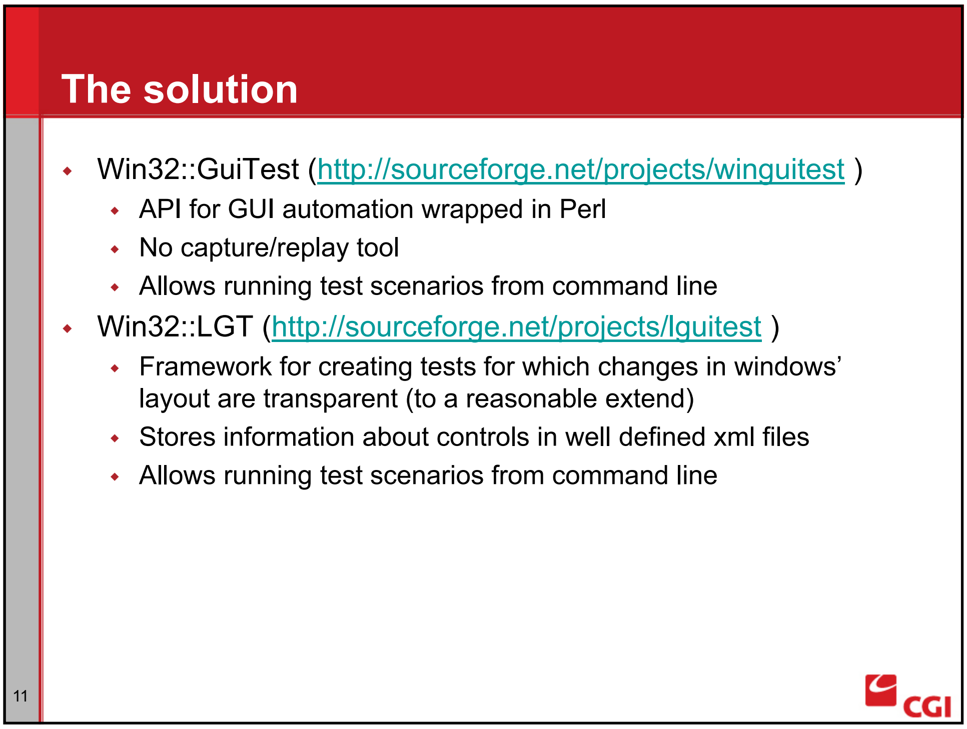
9



The solution

- Billing test automation
  - Find data for test scenarios – done by sqls and Perl
  - Run GUI scenarios – Win32::LGT (using results from previous steps)
  - For scenarios from previous steps generate call data records – custom Perl scripts
  - Rate the usage
  - Run the billing
  - Verify results – LRpt

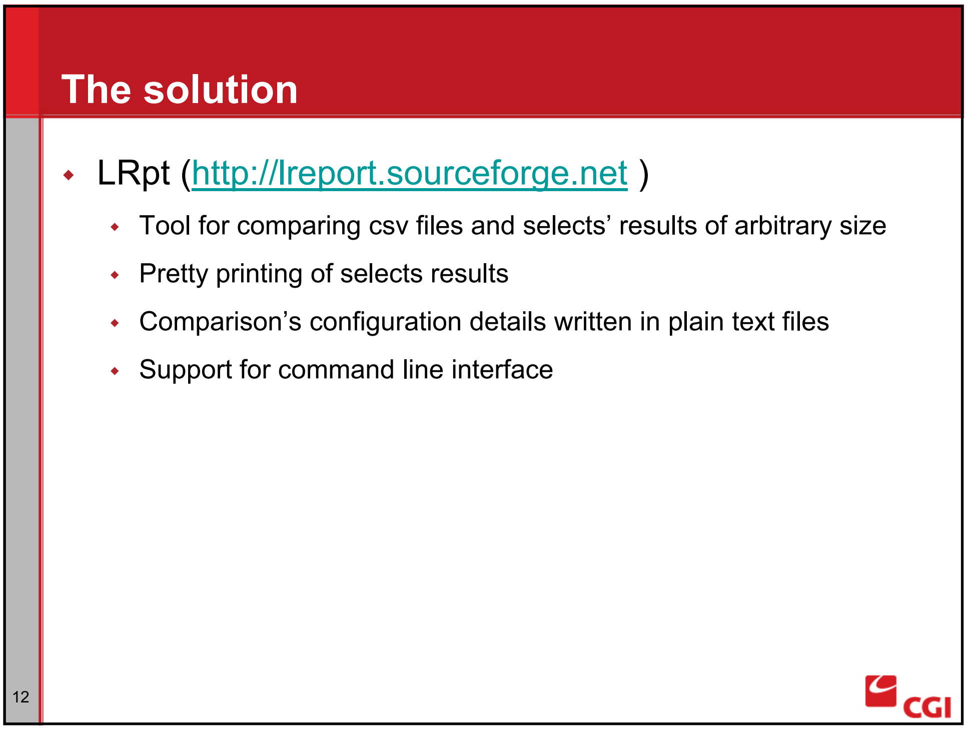
10



The solution

- Win32::GuiTest (http://sourceforge.net/projects/winguitest)
  - API for GUI automation wrapped in Perl
  - No capture/replay tool
  - Allows running test scenarios from command line
- Win32::LGT (http://sourceforge.net/projects/lguitest)
  - Framework for creating tests for which changes in windows' layout are transparent (to a reasonable extend)
  - Stores information about controls in well defined xml files
  - Allows running test scenarios from command line

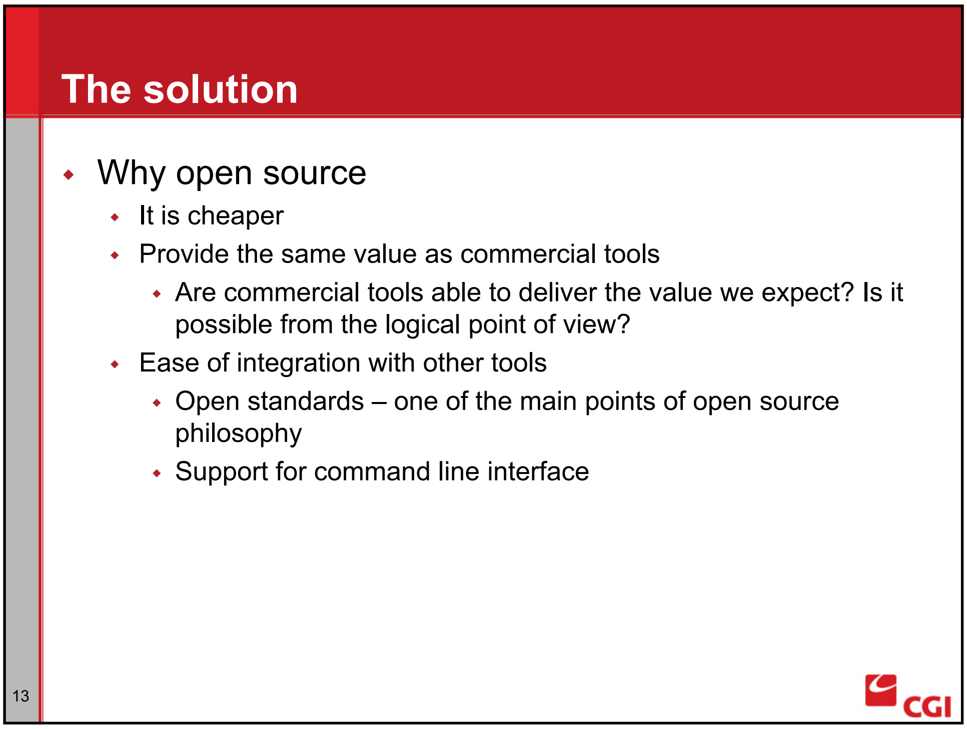
11



The solution

- LRpt (http://lreport.sourceforge.net)
  - Tool for comparing csv files and selects' results of arbitrary size
  - Pretty printing of selects results
  - Comparison's configuration details written in plain text files
  - Support for command line interface

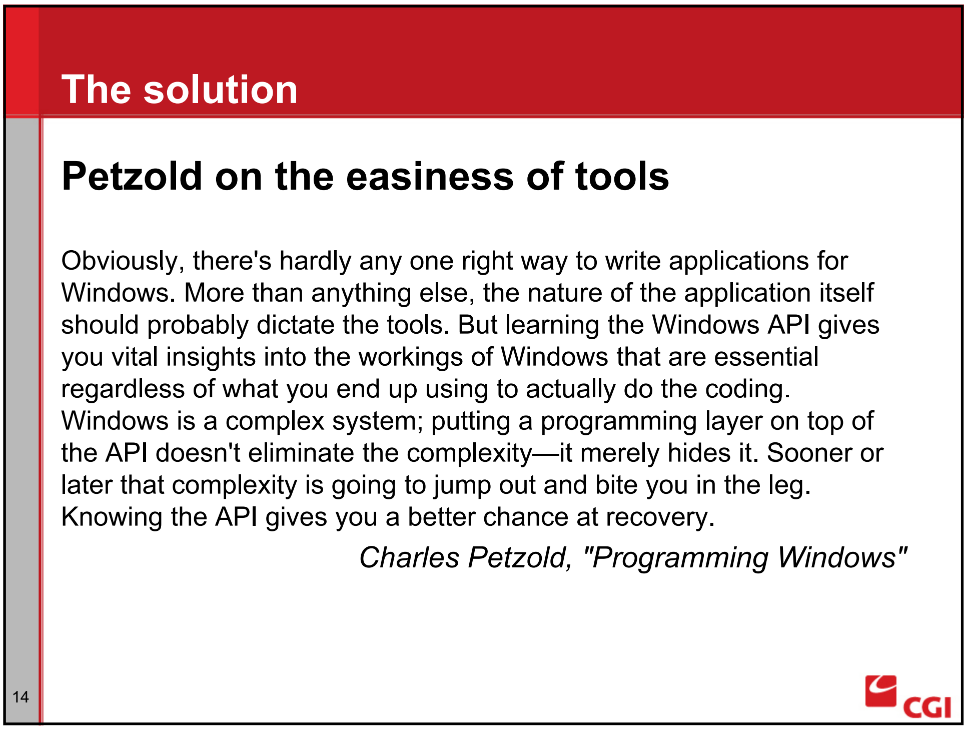
12



The solution

- Why open source
  - It is cheaper
  - Provide the same value as commercial tools
    - Are commercial tools able to deliver the value we expect? Is it possible from the logical point of view?
  - Ease of integration with other tools
    - Open standards – one of the main points of open source philosophy
    - Support for command line interface

13



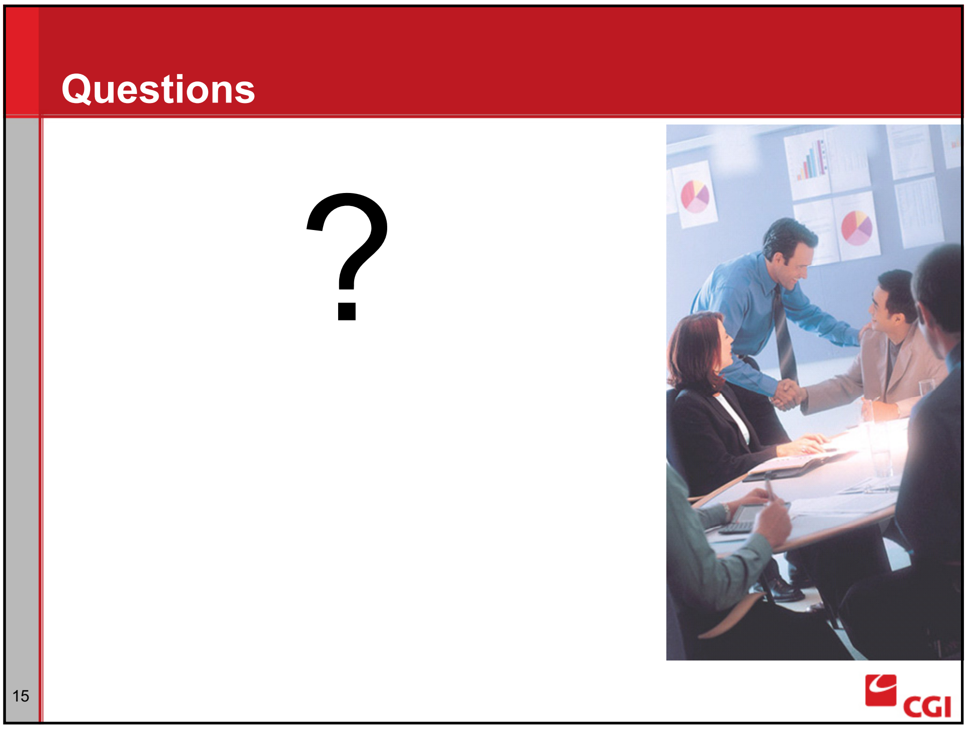
The solution

Petzold on the easiness of tools

Obviously, there's hardly any one right way to write applications for Windows. More than anything else, the nature of the application itself should probably dictate the tools. But learning the Windows API gives you vital insights into the workings of Windows that are essential regardless of what you end up using to actually do the coding. Windows is a complex system; putting a programming layer on top of the API doesn't eliminate the complexity—it merely hides it. Sooner or later that complexity is going to jump out and bite you in the leg. Knowing the API gives you a better chance at recovery.

Charles Petzold, "Programming Windows"

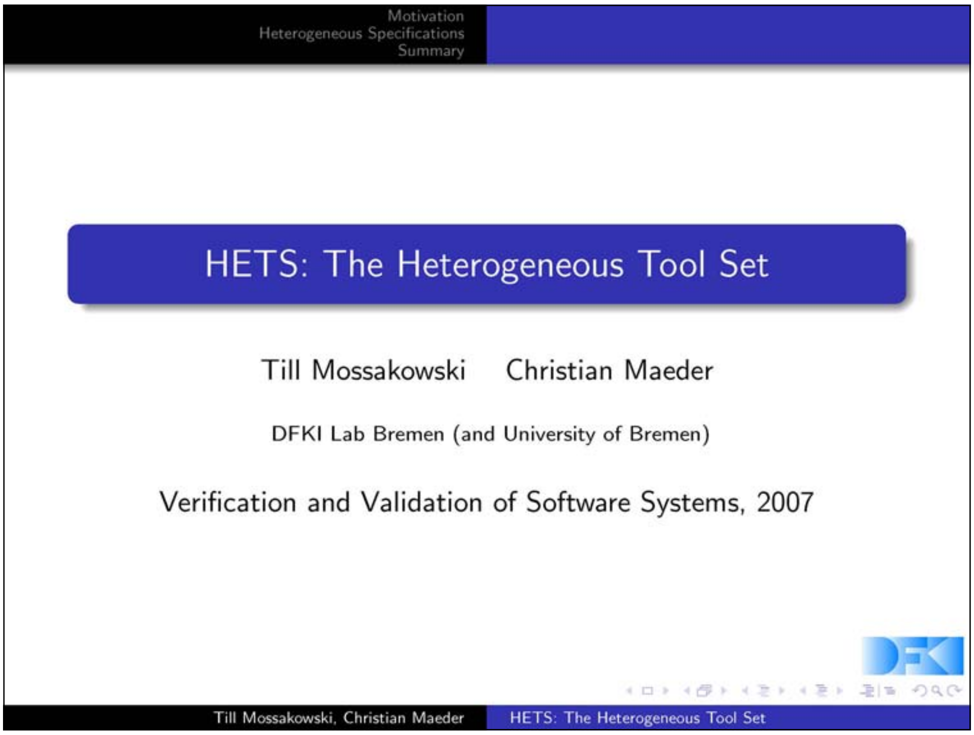
14



Questions

?

15



Motivation
Heterogeneous Specifications
Summary

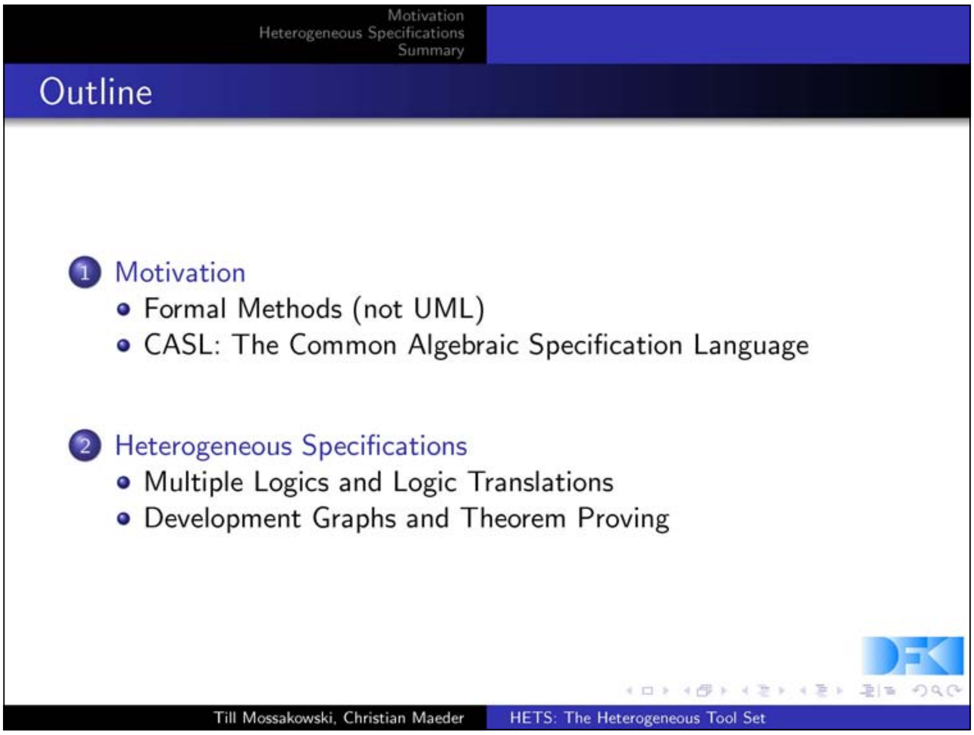
HETS: The Heterogeneous Tool Set

Till Mossakowski Christian Maeder

DFKI Lab Bremen (and University of Bremen)

Verification and Validation of Software Systems, 2007

Till Mossakowski, Christian Maeder HETS: The Heterogeneous Tool Set

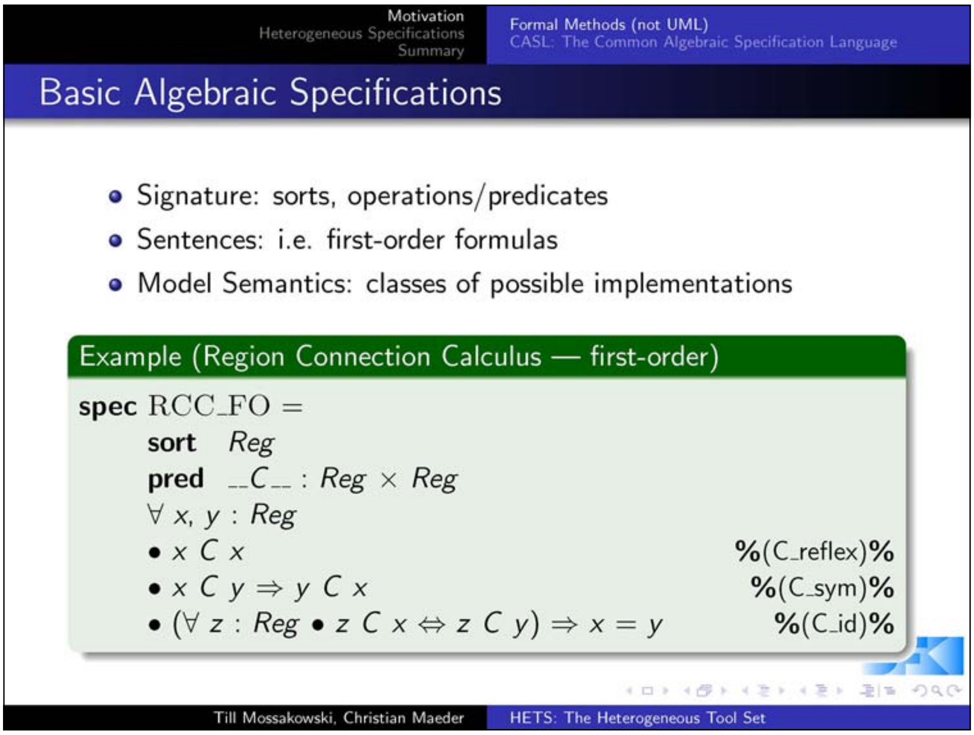


Motivation
Heterogeneous Specifications
Summary

Outline

1. Motivation
   - Formal Methods (not UML)
   - CASL: The Common Algebraic Specification Language
2. Heterogeneous Specifications
   - Multiple Logics and Logic Translations
   - Development Graphs and Theorem Proving

Till Mossakowski, Christian Maeder HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Formal Methods (not UML)
CASL: The Common Algebraic Specification Language

Basic Algebraic Specifications

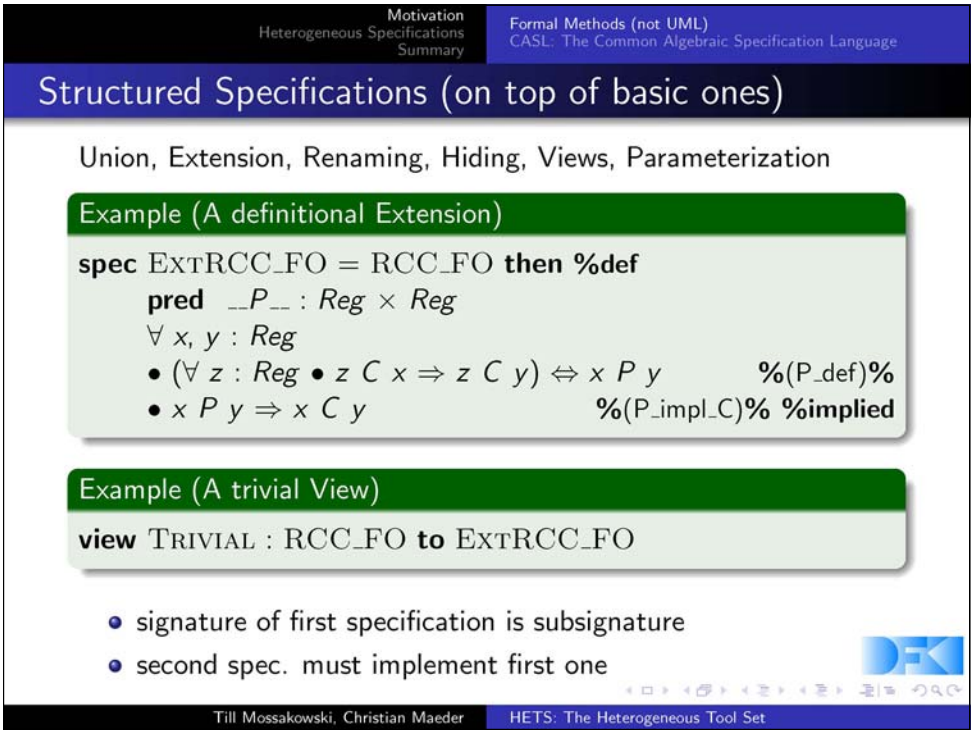
- Signature: sorts, operations/predicates
- Sentences: i.e. first-order formulas
- Model Semantics: classes of possible implementations

Example (Region Connection Calculus — first-order)

```
spec RCC_FO =
     sort  Reg
     pred  __C__ : Reg × Reg
     ∀ x, y : Reg
     • x C x                                           %(C_reflex)%
     • x C y ⇒ y C x                                   %(C_sym)%
     • (∀ z : Reg • z C x ⇔ z C y) ⇒ x = y             %(C_id)%
```

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Formal Methods (not UML)
CASL: The Common Algebraic Specification Language

Structured Specifications (on top of basic ones)

Union, Extension, Renaming, Hiding, Views, Parameterization

Example (A definitional Extension)

```
spec ExtRCC_FO = RCC_FO then %def
     pred  __P__ : Reg × Reg
     ∀ x, y : Reg
     • (∀ z : Reg • z C x ⇒ z C y) ⇔ x P y            %(P_def)%
     • x P y ⇒ x C y                         %(P_impl_C)% %implied
```

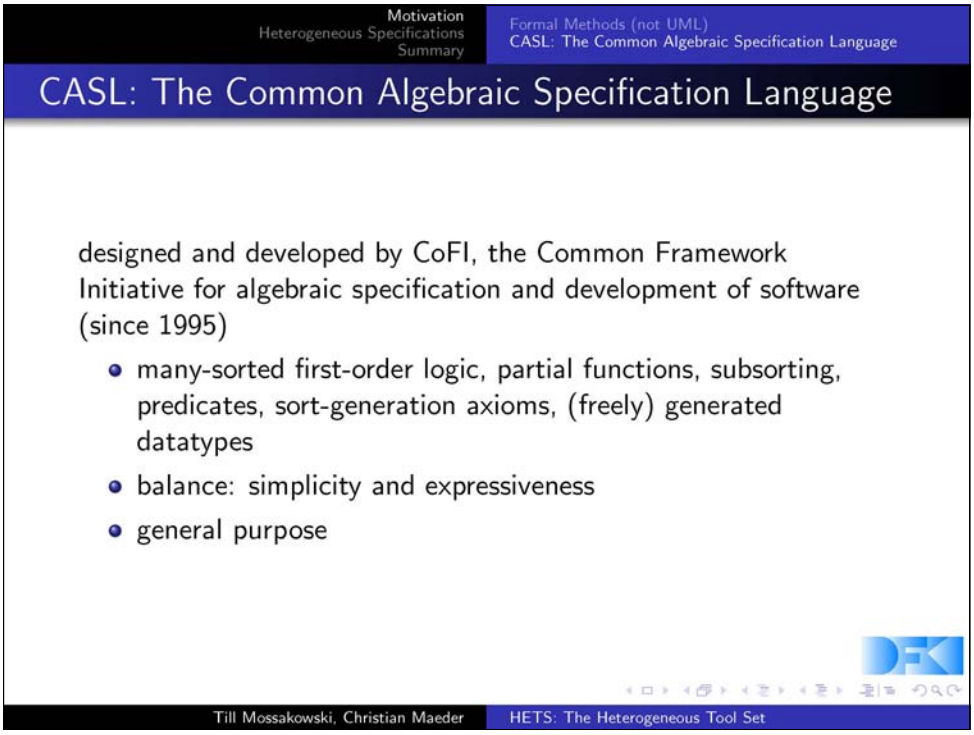
Example (A trivial View)

```
view Trivial : RCC_FO to ExtRCC_FO
```

- signature of first specification is subsignature
- second spec. must implement first one

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Formal Methods (not UML)
CASL: The Common Algebraic Specification Language

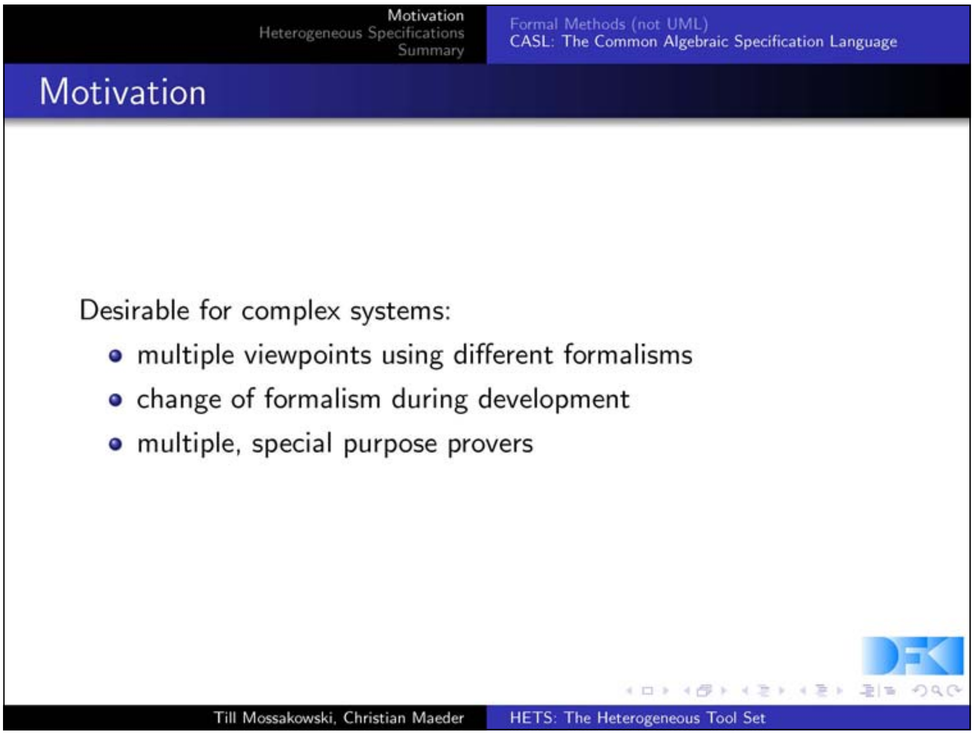
CASL: The Common Algebraic Specification Language

designed and developed by CoFI, the Common Framework Initiative for algebraic specification and development of software (since 1995)

- many-sorted first-order logic, partial functions, subsorting, predicates, sort-generation axioms, (freely) generated datatypes
- balance: simplicity and expressiveness
- general purpose

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Formal Methods (not UML)
CASL: The Common Algebraic Specification Language

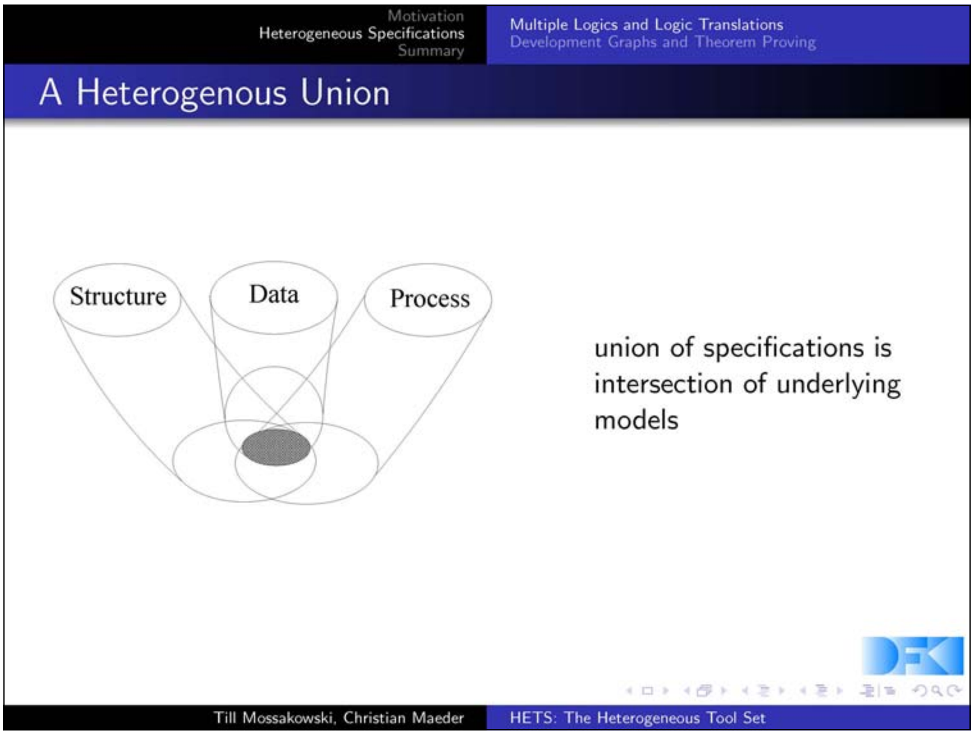
Motivation

Desirable for complex systems:

- multiple viewpoints using different formalisms
- change of formalism during development
- multiple, special purpose provers

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

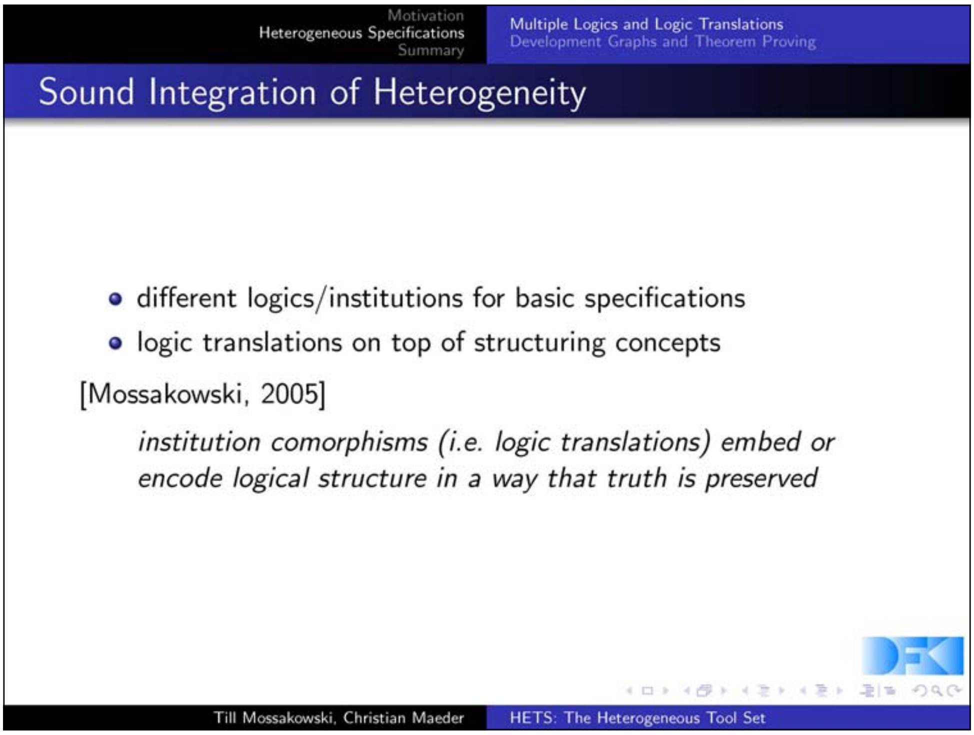
Multiple Logics and Logic Translations
Development Graphs and Theorem Proving

A Heterogenous Union

union of specifications is intersection of underlying models

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Multiple Logics and Logic Translations
Development Graphs and Theorem Proving

Sound Integration of Heterogeneity

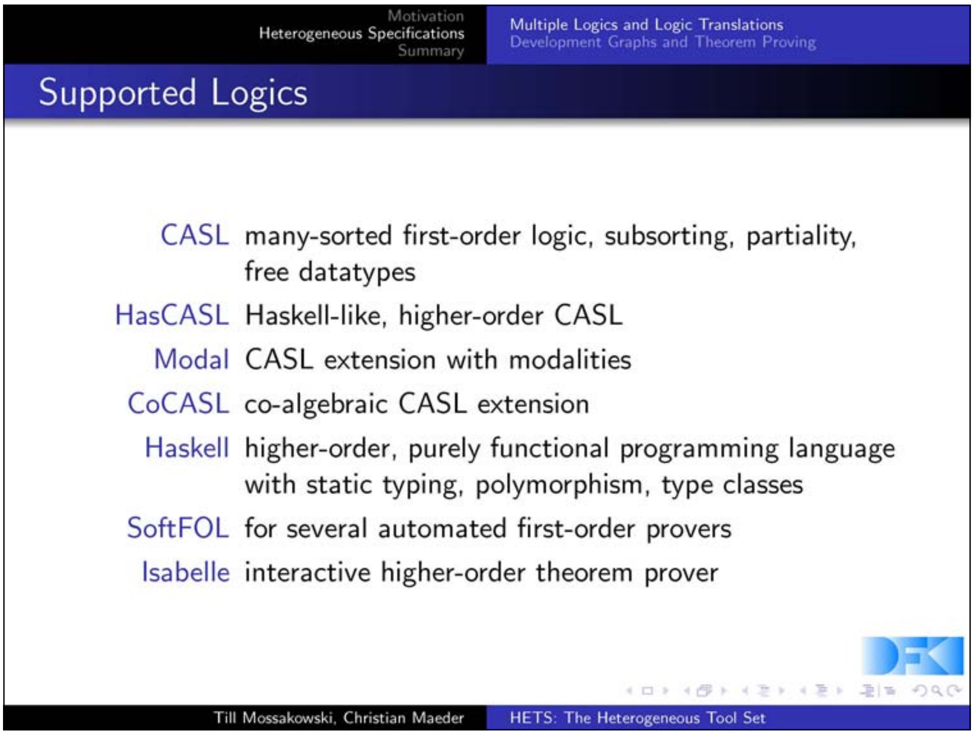
- different logics/institutions for basic specifications
- logic translations on top of structuring concepts

[Mossakowski, 2005]

> *institution comorphisms (i.e. logic translations) embed or encode logical structure in a way that truth is preserved*

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

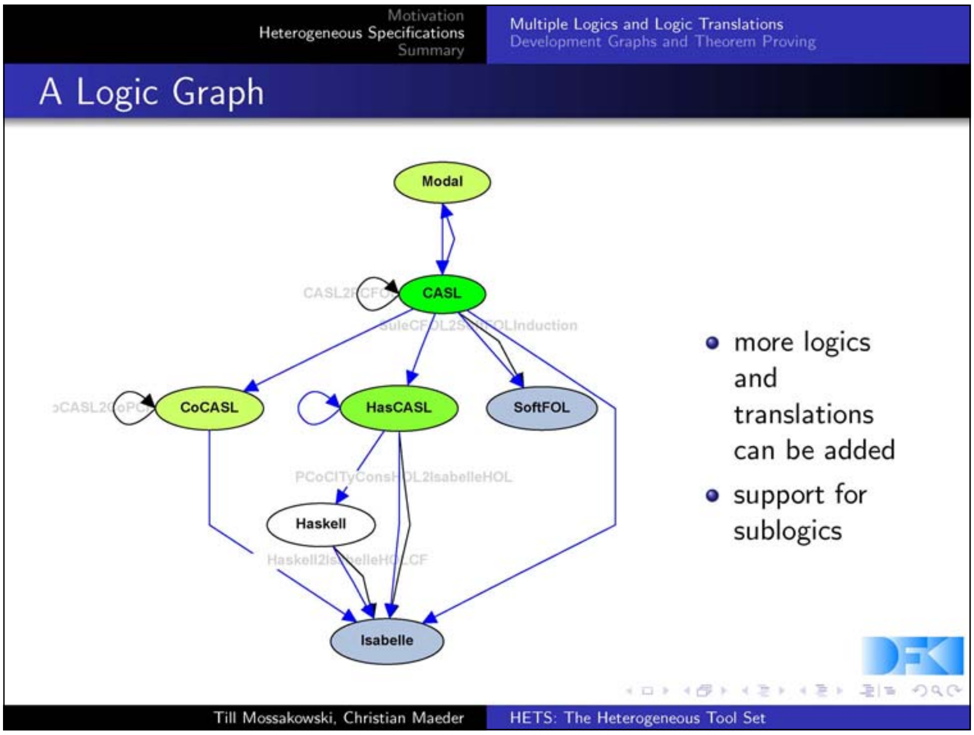
Multiple Logics and Logic Translations
Development Graphs and Theorem Proving

Supported Logics

CASL	many-sorted first-order logic, subsorting, partiality, free datatypes
HasCASL	Haskell-like, higher-order CASL
Modal	CASL extension with modalities
CoCASL	co-algebraic CASL extension
Haskell	higher-order, purely functional programming language with static typing, polymorphism, type classes
SoftFOL	for several automated first-order provers
Isabelle	interactive higher-order theorem prover

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

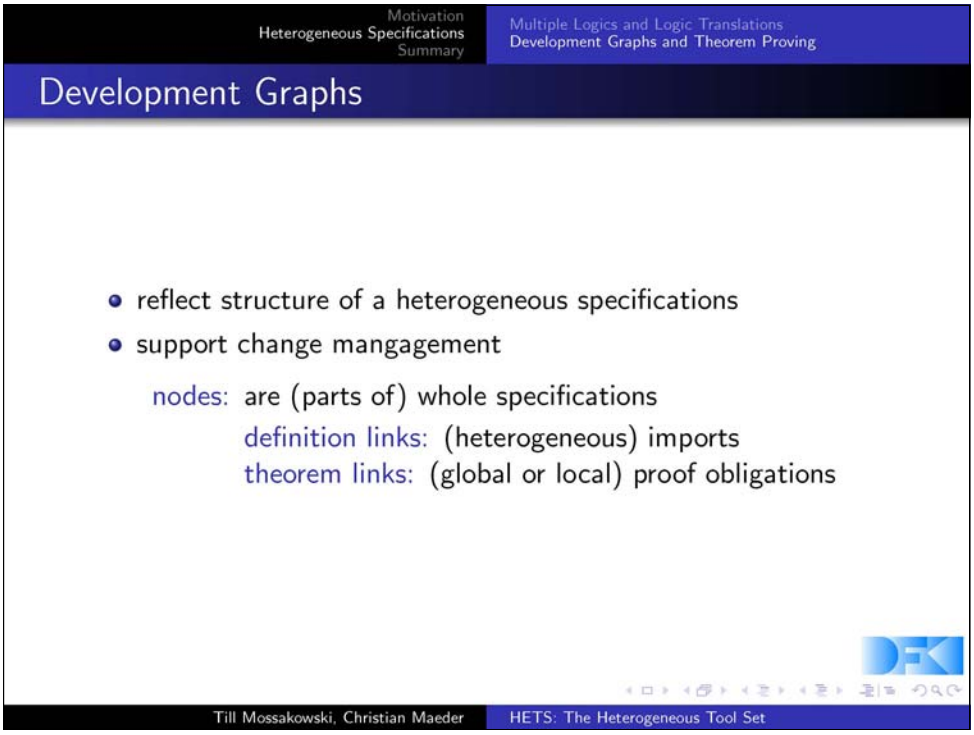
Multiple Logics and Logic Translations
Development Graphs and Theorem Proving

A Logic Graph

- more logics and translations can be added
- support for sublogics

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Multiple Logics and Logic Translations
Development Graphs and Theorem Proving

Development Graphs

- reflect structure of a heterogeneous specifications
- support change mangagement

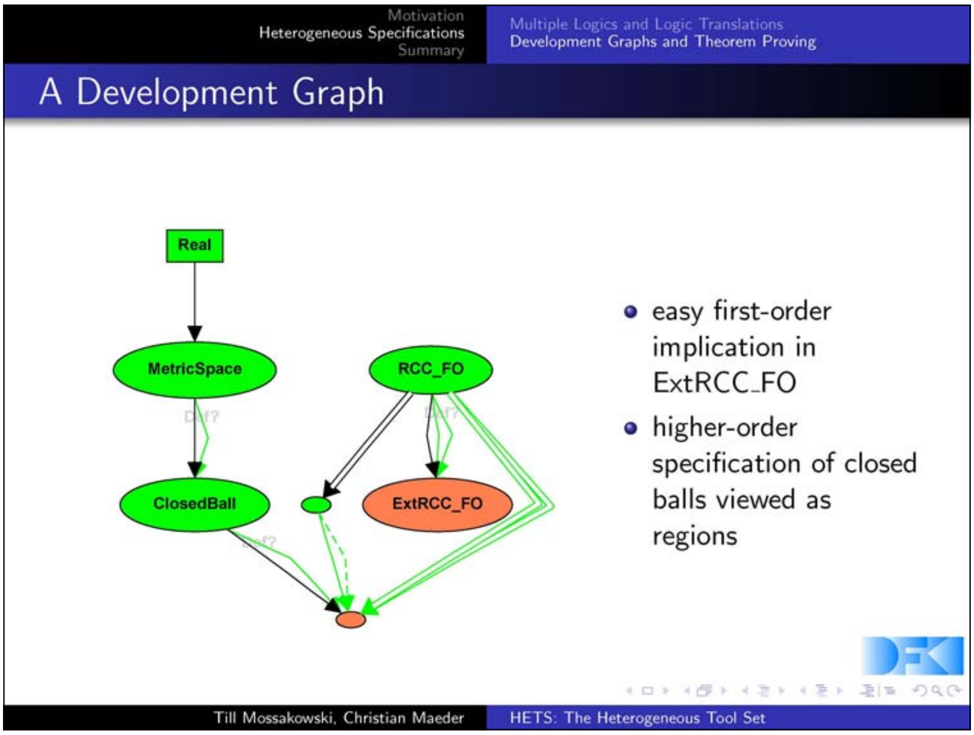
nodes: are (parts of) whole specifications

definition links: (heterogeneous) imports

theorem links: (global or local) proof obligations

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set



Motivation
Heterogeneous Specifications
Summary

Multiple Logics and Logic Translations
Development Graphs and Theorem Proving

A Development Graph

- easy first-order implication in ExtRCC_FO
- higher-order specification of closed balls viewed as regions

Till Mossakowski, Christian Maeder

HETS: The Heterogeneous Tool Set

Proof Calculus

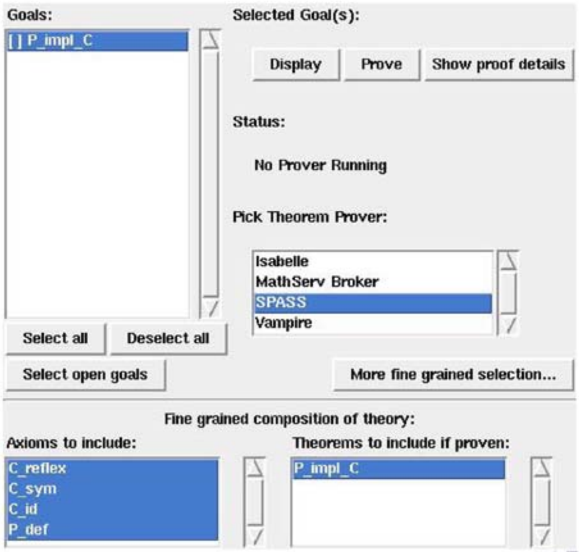
[Mossakowski, 2005]

Theorem

The proof calculus for heterogeneous development graphs is sound and complete relative to an oracle checking conservative extensions.

- decompose global theorem links automatically
- choose specific provers for local proof goals
- use model checkers to construct models

Prover Selection Dialog



Goals:

[ ] P_impl_C

Selected Goal(s):

Status:

No Prover Running

Pick Theorem Prover:

Isabelle
MathServ Broker
SPASS
Vampire

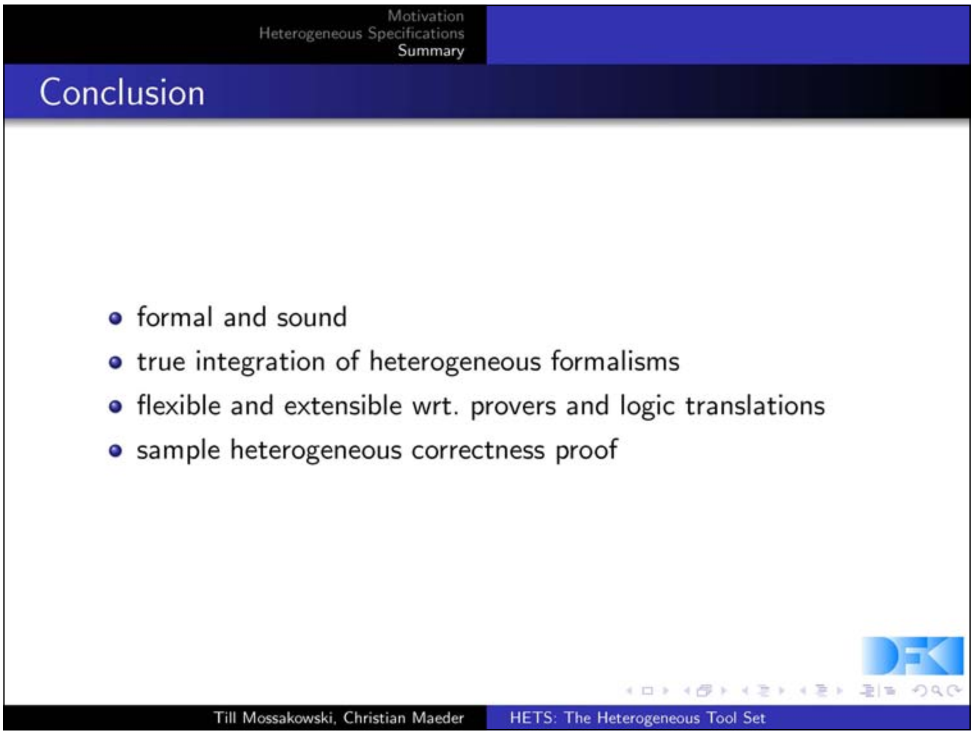
Fine grained composition of theory:

Axioms to include:

C_reflex
C_sym
C_id
P_def

Theorems to include if proven:

P_impl_C

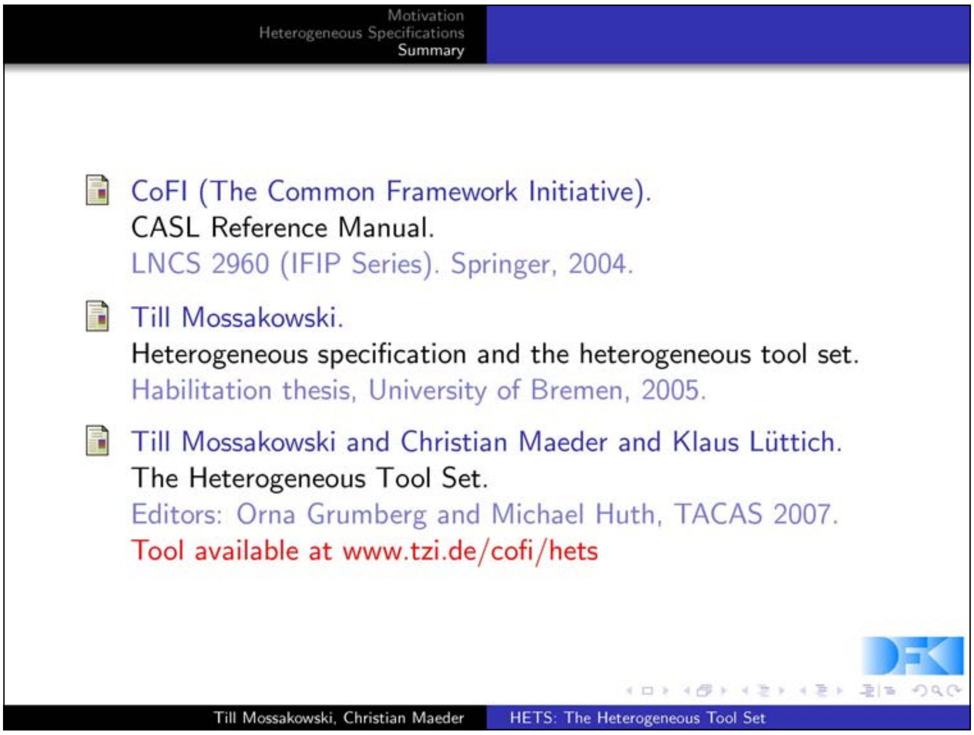


Motivation
Heterogeneous Specifications
Summary

Conclusion

- formal and sound
- true integration of heterogeneous formalisms
- flexible and extensible wrt. provers and logic translations
- sample heterogeneous correctness proof

Till Mossakowski, Christian Maeder | HETS: The Heterogeneous Tool Set



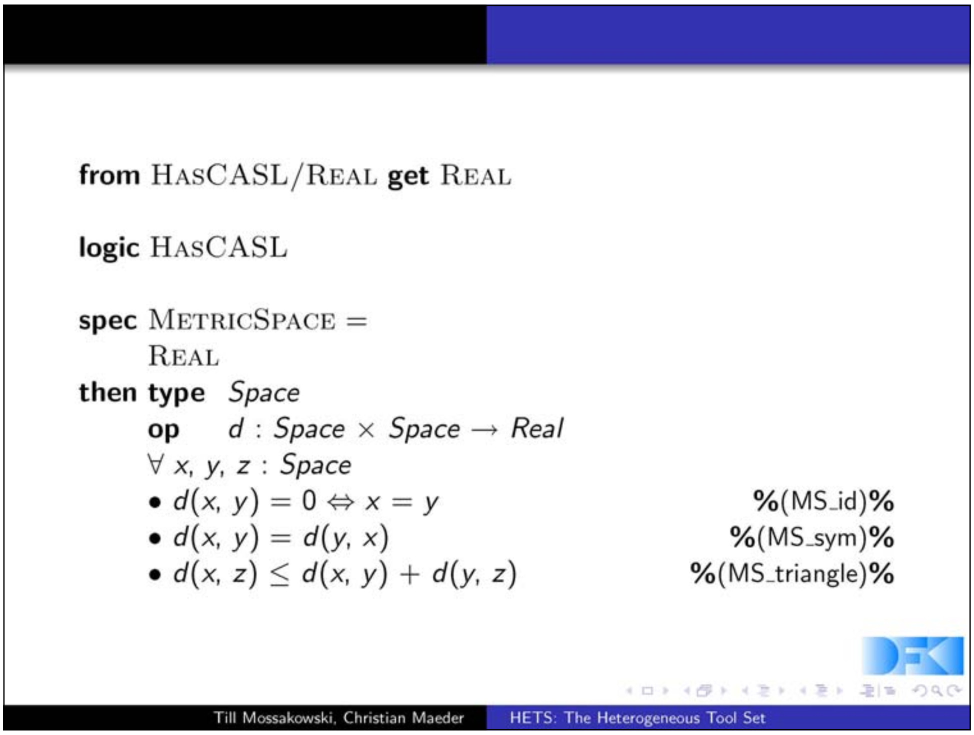
Motivation
Heterogeneous Specifications
Summary

CoFI (The Common Framework Initiative).
CASL Reference Manual.
LNCS 2960 (IFIP Series). Springer, 2004.

Till Mossakowski.
Heterogeneous specification and the heterogeneous tool set.
Habilitation thesis, University of Bremen, 2005.

Till Mossakowski and Christian Maeder and Klaus Lüttich.
The Heterogeneous Tool Set.
Editors: Orna Grumberg and Michael Huth, TACAS 2007.
Tool available at www.tzi.de/cofi/hets

Till Mossakowski, Christian Maeder | HETS: The Heterogeneous Tool Set

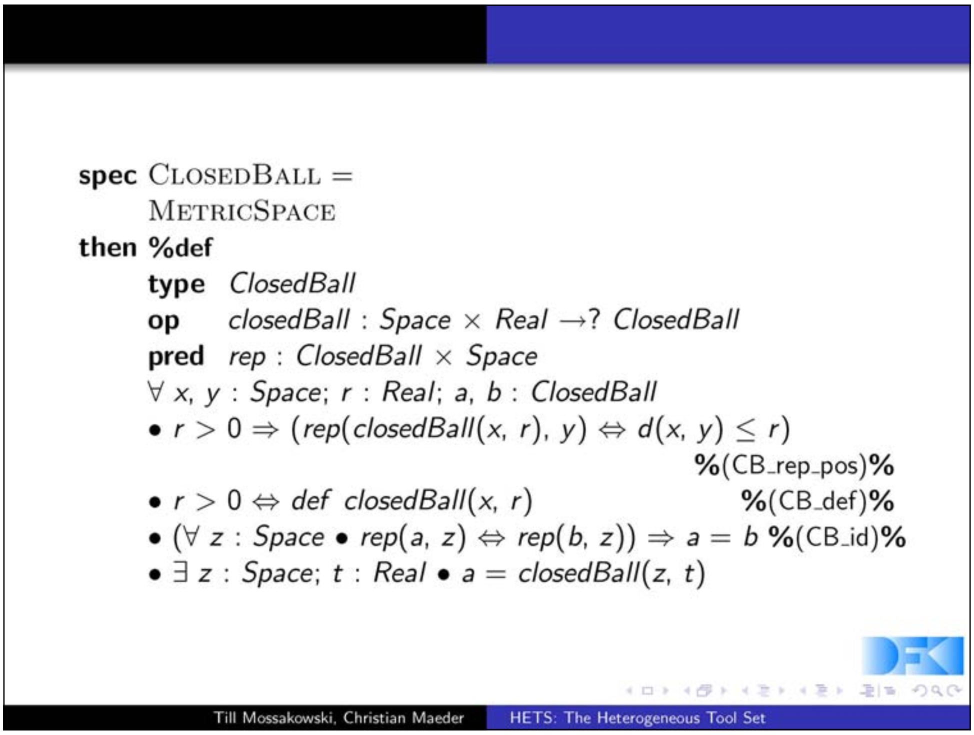


from HasCASL/Real **get** Real

logic HasCASL

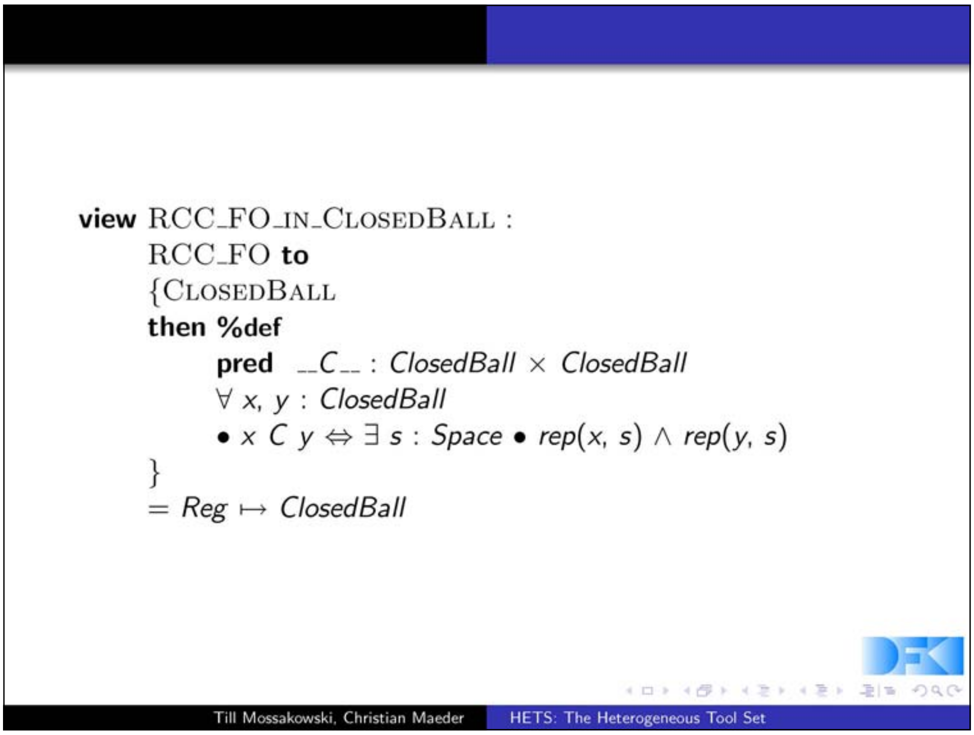
spec MetricSpace =
Real
then type *Space*
op $d : Space \times Space \to Real$
$\forall\, x, y, z : Space$
- $d(x, y) = 0 \Leftrightarrow x = y$ **%**(MS_id)**%**
- $d(x, y) = d(y, x)$ **%**(MS_sym)**%**
- $d(x, z) \leq d(x, y) + d(y, z)$ **%**(MS_triangle)**%**

Till Mossakowski, Christian Maeder HETS: The Heterogeneous Tool Set



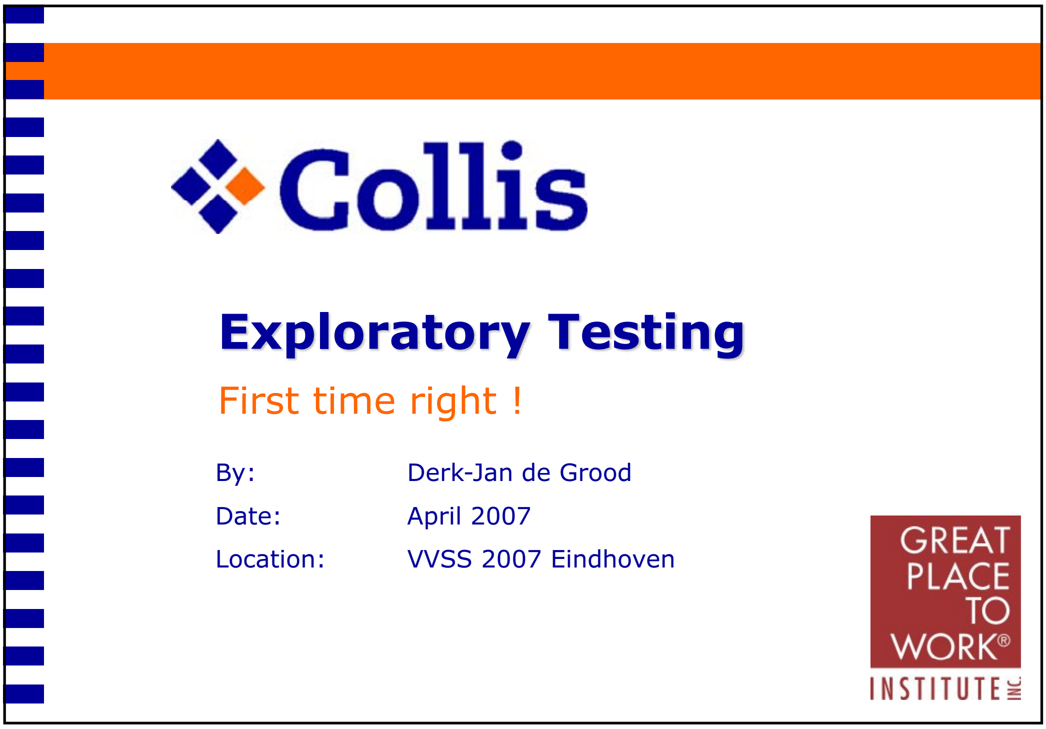
spec ClosedBall =
MetricSpace
then %def
type *ClosedBall*
op $closedBall : Space \times Real \to ?\ ClosedBall$
pred $rep : ClosedBall \times Space$
$\forall\, x, y : Space;\ r : Real;\ a, b : ClosedBall$
- $r > 0 \Rightarrow (rep(closedBall(x, r), y) \Leftrightarrow d(x, y) \leq r)$
%(CB_rep_pos)**%**
- $r > 0 \Leftrightarrow def\ closedBall(x, r)$ **%**(CB_def)**%**
- $(\forall\, z : Space \bullet rep(a, z) \Leftrightarrow rep(b, z)) \Rightarrow a = b$ **%**(CB_id)**%**
- $\exists\, z : Space;\ t : Real \bullet a = closedBall(z, t)$

Till Mossakowski, Christian Maeder HETS: The Heterogeneous Tool Set



```
view RCC_FO_IN_CLOSEDBALL :
       RCC_FO to
       {CLOSEDBALL
       then %def
              pred   __C__ : ClosedBall × ClosedBall
              ∀ x, y : ClosedBall
              • x C y ⇔ ∃ s : Space • rep(x, s) ∧ rep(y, s)
       }
       = Reg ↦ ClosedBall
```

Till Mossakowski, Christian Maeder   HETS: The Heterogeneous Tool Set



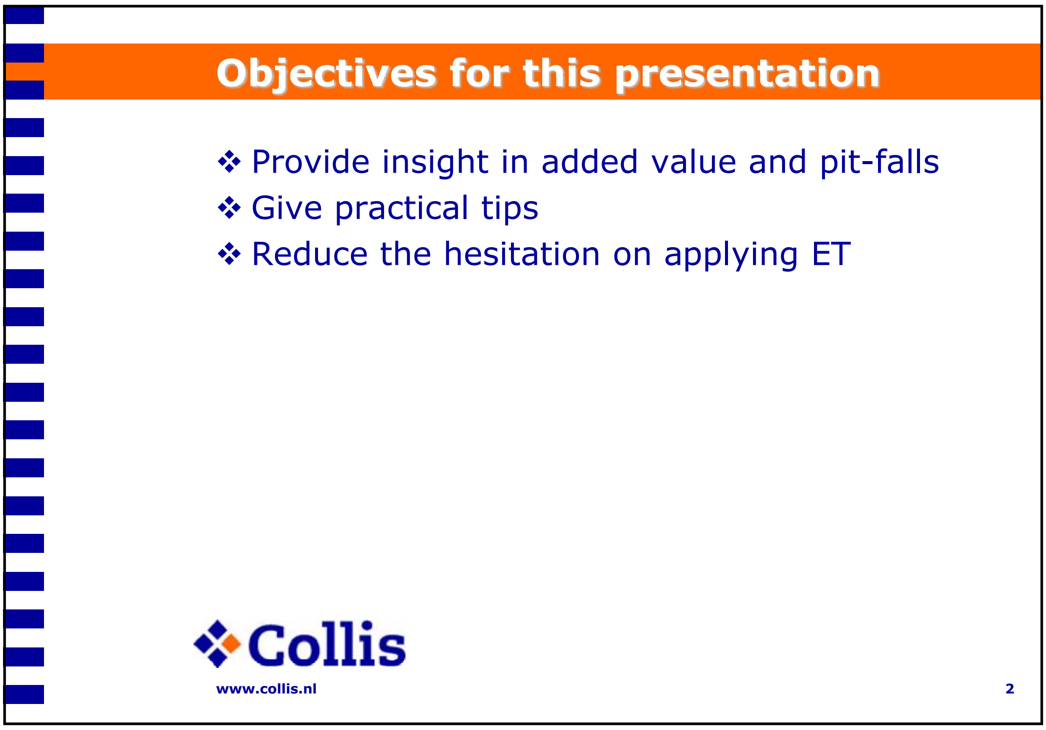
Collis

Exploratory Testing

First time right !

By:	Derk-Jan de Grood
Date:	April 2007
Location:	VVSS 2007 Eindhoven

GREAT PLACE TO WORK® INSTITUTE INC.



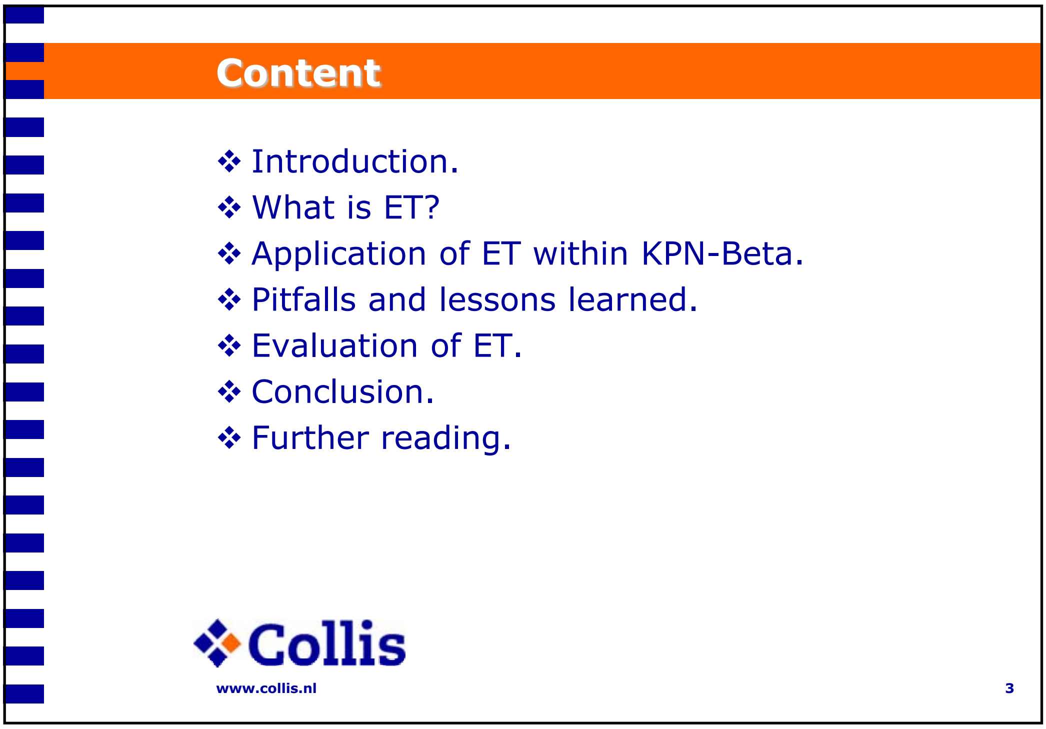
Objectives for this presentation

- Provide insight in added value and pit-falls
- Give practical tips
- Reduce the hesitation on applying ET

Collis

www.collis.nl

2

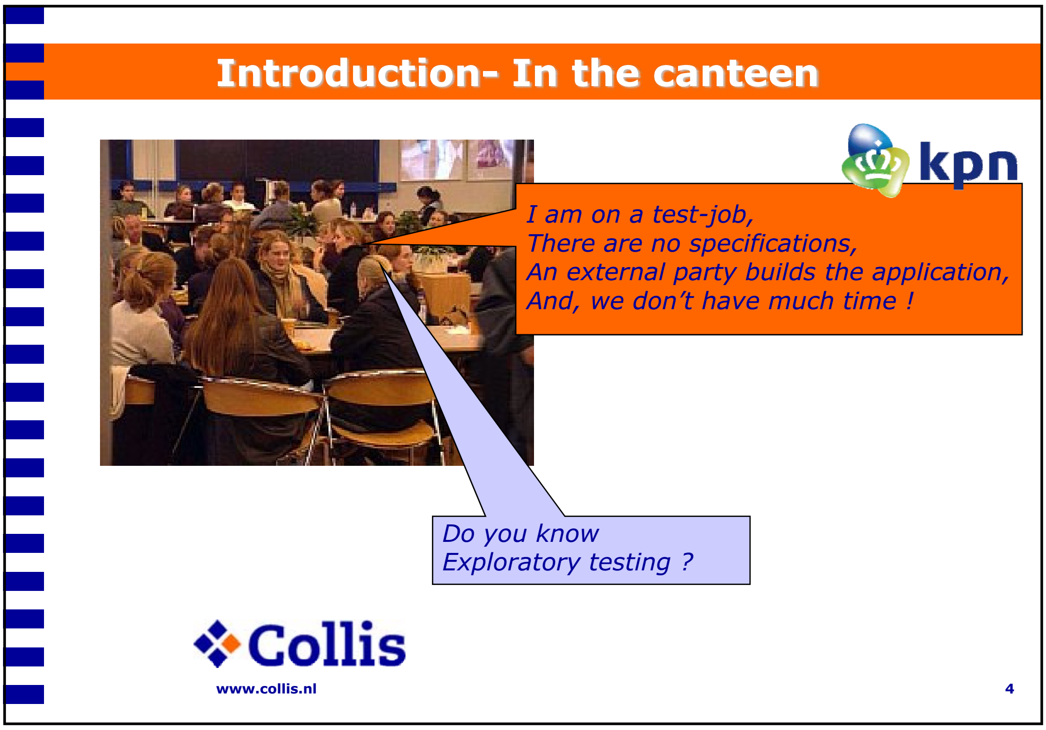


Content

- ❖ Introduction.
- ❖ What is ET?
- ❖ Application of ET within KPN-Beta.
- ❖ Pitfalls and lessons learned.
- ❖ Evaluation of ET.
- ❖ Conclusion.
- ❖ Further reading.

Collis
www.collis.nl

3



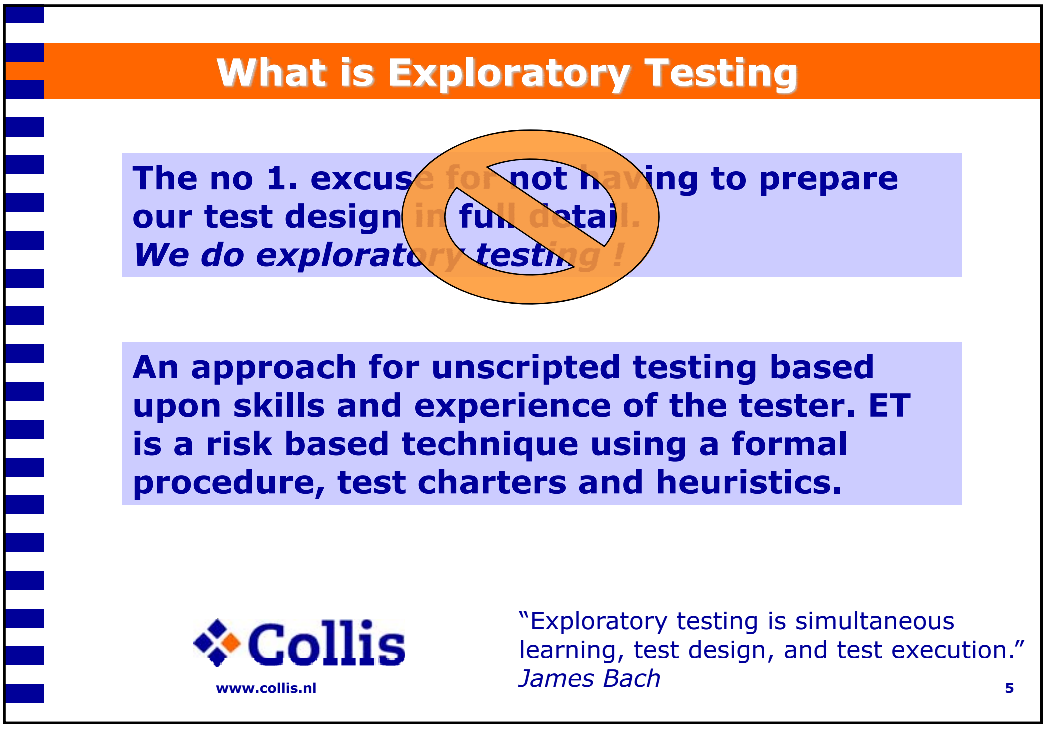
Introduction- In the canteen

I am on a test-job,
There are no specifications,
An external party builds the application,
And, we don't have much time !

Do you know
Exploratory testing ?

Collis
www.collis.nl

4



What is Exploratory Testing

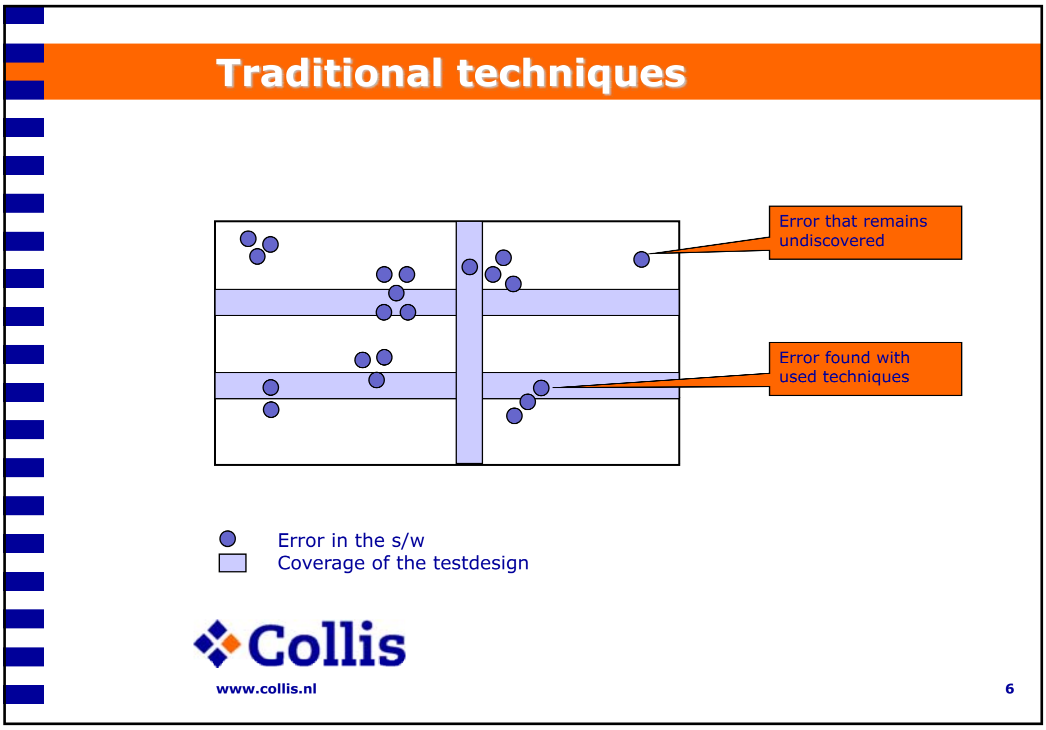
The no 1. excuse for not having to prepare our test design in full detail.
We do exploratory testing !

An approach for unscripted testing based upon skills and experience of the tester. ET is a risk based technique using a formal procedure, test charters and heuristics.

“Exploratory testing is simultaneous learning, test design, and test execution.”
James Bach

www.collis.nl

5



Traditional techniques

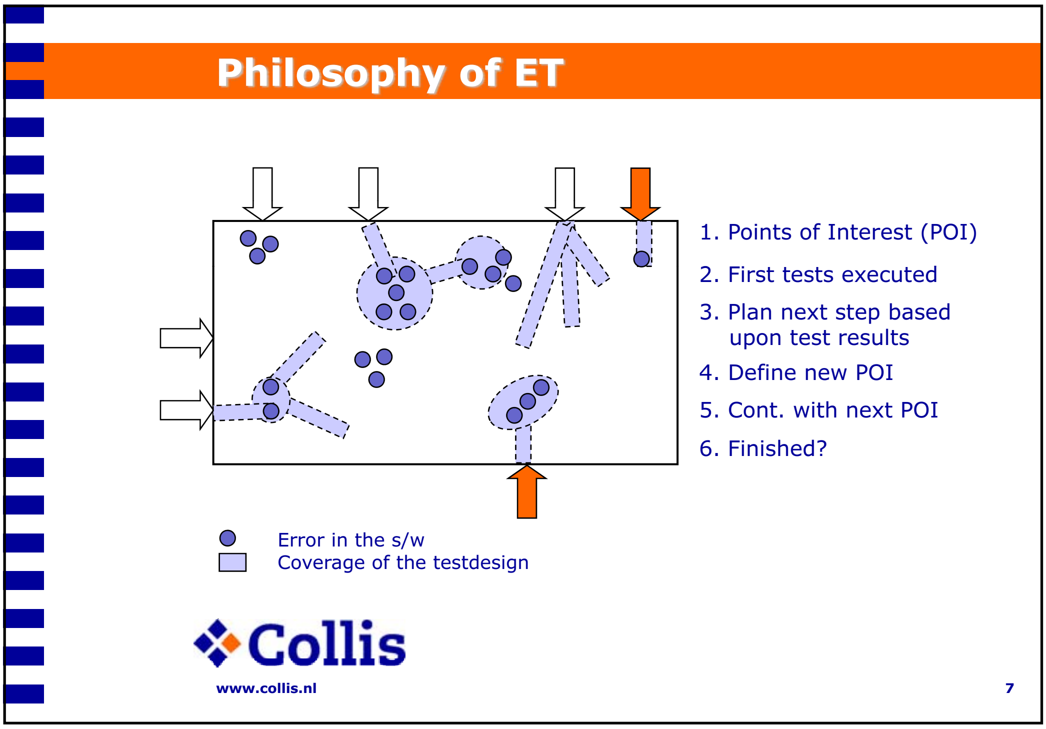
Error that remains undiscovered

Error found with used techniques

● Error in the s/w
▭ Coverage of the testdesign

www.collis.nl

6



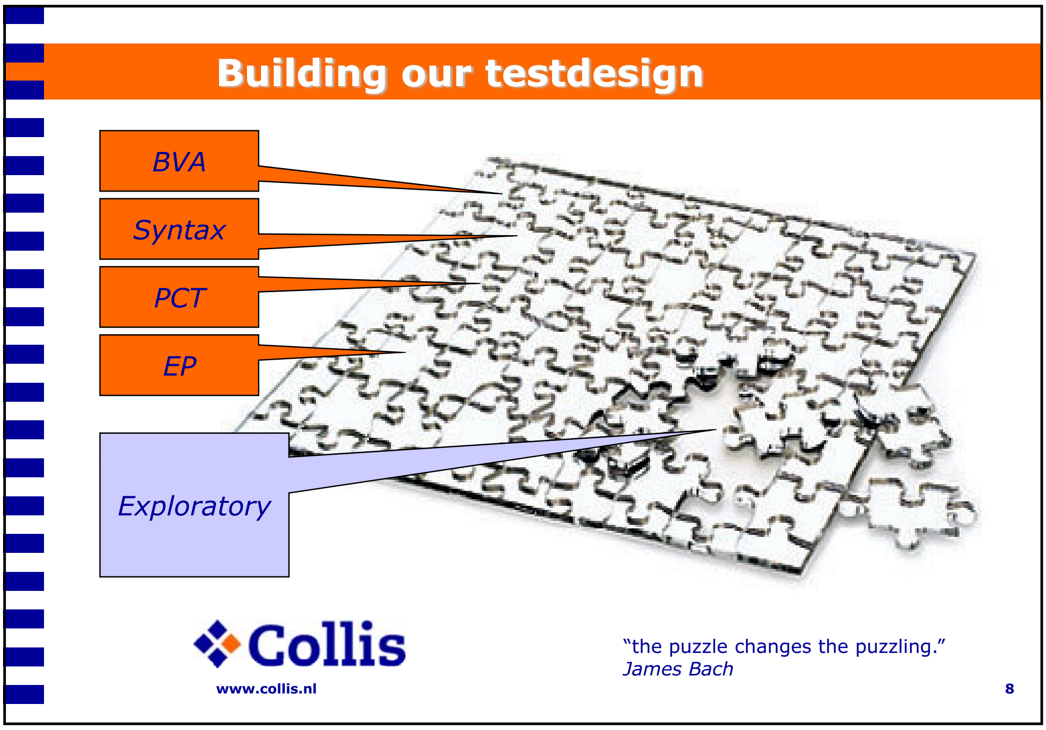
Philosophy of ET

1. Points of Interest (POI)
2. First tests executed
3. Plan next step based upon test results
4. Define new POI
5. Cont. with next POI
6. Finished?

● Error in the s/w
▭ Coverage of the testdesign

Collis
www.collis.nl

7

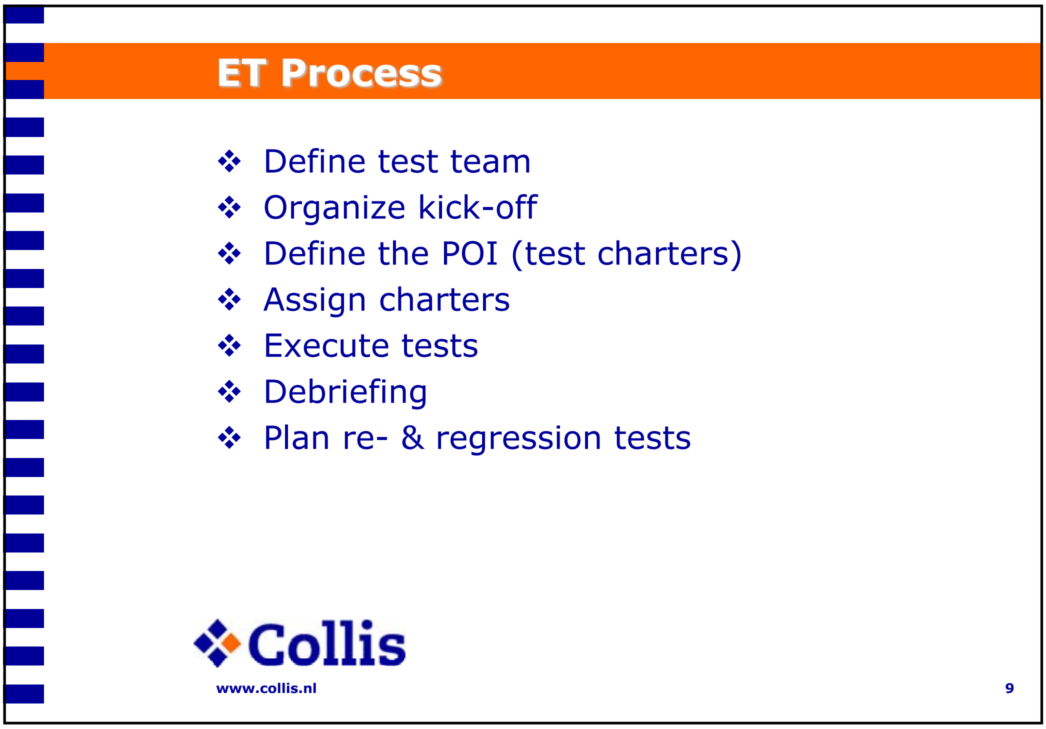


Building our testdesign

Collis
www.collis.nl

"the puzzle changes the puzzling."
James Bach

8



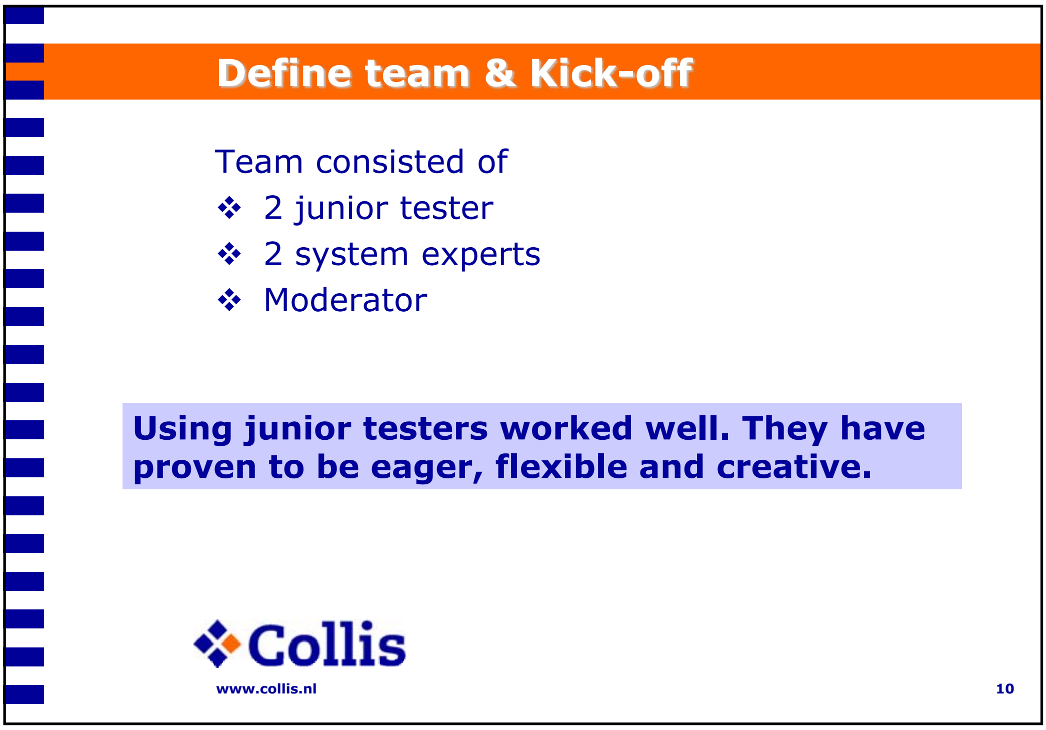
ET Process

- Define test team
- Organize kick-off
- Define the POI (test charters)
- Assign charters
- Execute tests
- Debriefing
- Plan re- & regression tests

Collis

www.collis.nl

9



Define team & Kick-off

Team consisted of

- 2 junior tester
- 2 system experts
- Moderator

Using junior testers worked well. They have proven to be eager, flexible and creative.

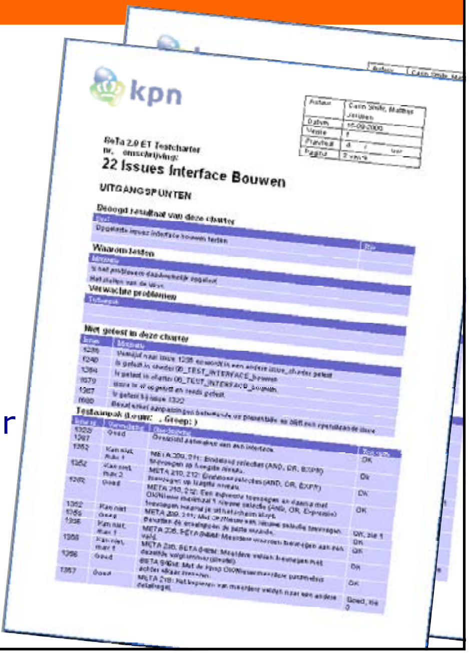
Collis

www.collis.nl

10

Define Charter

- ❖ Charter ID
- ❖ Priority
- ❖ Estimated time
- ❖ Aimed result
- ❖ Why should we test this?
- ❖ Expected problems
- ❖ Not included in this Charter
- ❖ Conclusion



Auteur	[illegible]
Datum	[illegible]
Versie	1
Prioriteit	[illegible]
Pagina	[illegible]

ReTa 2.0 ET Testcharter
nr. omschrijving:

22 Issues Interface Bouwen

UITGANGSPUNTEN

Beoogd resultaat van deze charter

Waarom testen

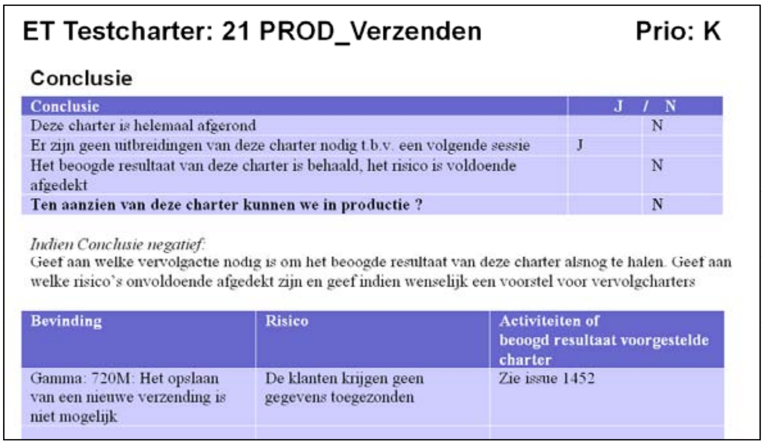
Verwachte problemen

Niet getest in deze charter

Testaanpak (Lengte: , Groep:)

Collis
www.collis.nl

Charter- the conclusion



ET Testcharter: 21 PROD_Verzenden Prio: K

Conclusie

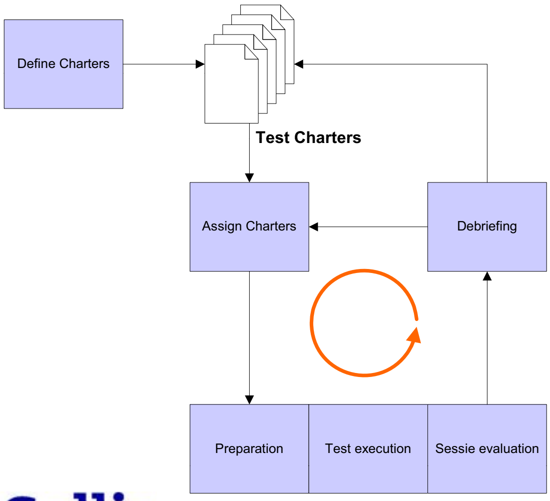
Conclusie	J	N
Deze charter is helemaal afgerond		N
Er zijn geen uitbreidingen van deze charter nodig t.b.v. een volgende sessie	J	
Het beoogde resultaat van deze charter is behaald, het risico is voldoende afgedekt		N
Ten aanzien van deze charter kunnen we in productie ?		N

Indien Conclusie negatief:
Geef aan welke vervolgactie nodig is om het beoogde resultaat van deze charter alsnog te halen. Geef aan welke risico's onvoldoende afgedekt zijn en geef indien wenselijk een voorstel voor vervolgcharters

Bevinding	Risico	Activiteiten of beoogd resultaat voorgestelde charter
Gamma: 720M: Het opslaan van een nieuwe verzending is niet mogelijk	De klanten krijgen geen gegevens toegezonden	Zie issue 1452

Collis
www.collis.nl

The iterative part of the process



Define Charters

Test Charters

Assign Charters

Debriefing

Preparation

Test execution

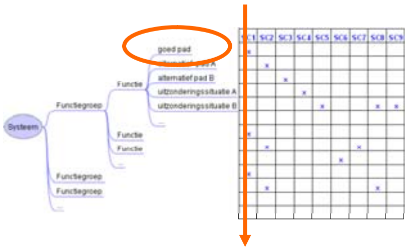
Sessie evaluation

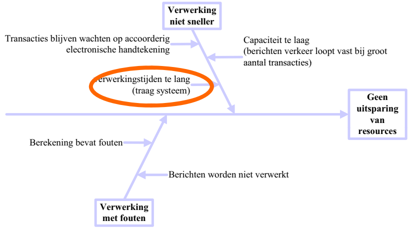
Collis

www.collis.nl

The content of the test charters

- How do you divide all functionality and risks that need to be tested over the test charters. Our experience with scenario and function based testing?





Verwerking niet sneller

Transacties blijven wachten op accoorderig electronische handtekening

Capaciteit te laag (berichten verkeer loopt vast bij groot aantal transacties)

erwerkingstijden te lang (traag systeem)

Geen uitsparing van resources

Berekening bevat fouten

Berichten worden niet verwerkt

Verwerking met fouten

Collis

www.collis.nl

Agility in test & process

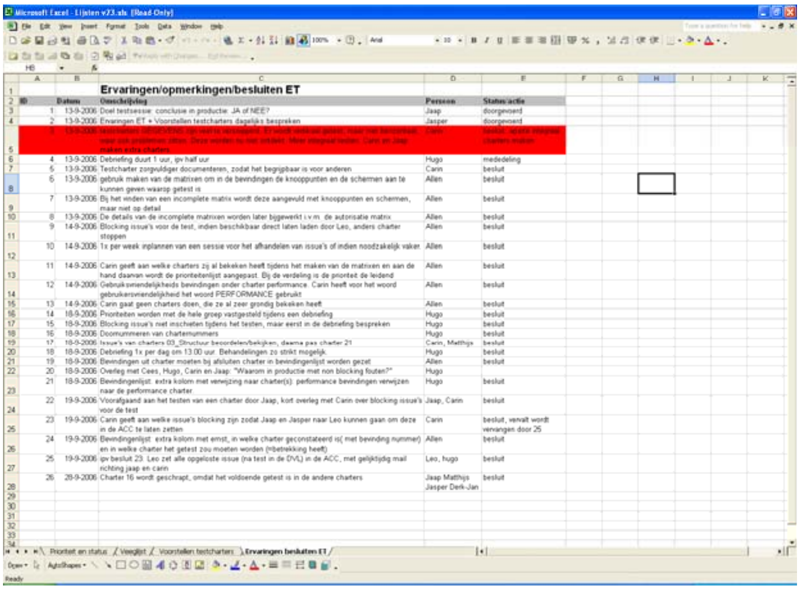
- ❖ When unacquainted with ET, expect the process to change over time.
- ❖ Debriefings were used to discus process.
- ❖ Make decisions.

Basic rule in agility: Decide on what you know at the time. Don't loose momentum by predicting what you might know tomorrow.

Collis

www.collis.nl

Observations & Decisions

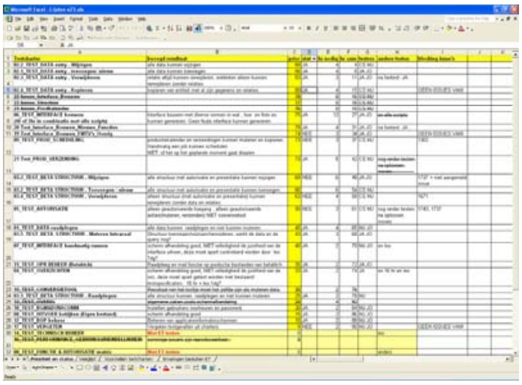


Ervaringen/opmerkingen/besluiten ET

ID	Datum	Omschrijving	Persoon	Status/actie
1	13-9-2006	Doel testsessie: conclusie in productie: JA of NEE?	Jaap	doorgevoerd
2	13-9-2006	Ervaringen ET + Voorstellen testcharters dagelijks bespreken	Jasper	doorgevoerd
3	13-9-2006	testcharters GEGEVENS zijn veel te versnipperd. Er wordt verticaal getest, maar niet horizontaal, waar ook problemen zitten. Deze worden nu niet ontdekt. Meer integraal testen. Carin en Jaap maken extra charters	Carin	besluit: aparte integraal charters maken
4	13-9-2006	Debriefing duurt 1 uur, ipv half uur	Hugo	mededeling
5	13-9-2006	Testcharter zorgvuldiger documenteren, zodat het begrijpbaar is voor anderen	Carin	besluit
6	13-9-2006	gebruik maken van de matrixen om in de bevindingen de knooppunten en de schermen aan te kunnen geven waarop getest is	Allen	besluit
7	13-9-2006	Bij het vinden van een incomplete matrix wordt deze aangevuld met knooppunten en schermen, maar niet op detail	Allen	besluit
8	13-9-2006	De details van de incomplete matrixen worden later bijgewerkt i.v.m. de autorisatie matrix	Allen	besluit
9	14-9-2006	Blocking issue's voor de test, indien beschikbaar direct laten laden door Leo, anders charter stoppen	Allen	besluit
10	14-9-2006	1x per week inplannen van een sessie voor het afhandelen van issue's of indien noodzakelijk vaker.	Allen	besluit
11	14-9-2006	Carin geeft aan welke charters zij al bekeken heeft tijdens het maken van de matrixen en aan de hand daarvan wordt de prioriteitenlijst aangepast. Bij de verdeling is de prioriteit de leidend	Allen	besluit
12	14-9-2006	Gebruiksvriendelijkheids bevindingen onder charter performance. Carin heeft voor het woord gebruikersvriendelijkheid het woord PERFORMANCE gebruikt	Allen	besluit
13	14-9-2006	Carin gaat geen charters doen, die ze al zeer grondig bekeken heeft	Allen	besluit
14	18-9-2006	Prioriteiten worden met de hele groep vastgesteld tijdens een debriefing	Hugo	besluit
15	18-9-2006	Blocking issue's niet inschieten tijdens het testen, maar eerst in de debriefing bespreken	Hugo	besluit
16	18-9-2006	Doornummeren van charternummers	Hugo	besluit
17	18-9-2006	Issue's van charters 03_Structuur beoordelen/bekijken, daarna pas charter 21	Carin, Matthijs	besluit
18	18-9-2006	Debriefing 1x per dag om 13.00 uur. Behandelingen zo strikt mogelijk.	Hugo	besluit
19	18-9-2006	Bevindingen uit charter moeten bij afsluiten charter in bevindingenlijst worden gezet	Allen	besluit
20	18-9-2006	Overleg met Cees, Hugo, Carin en Jaap: "Waarom in productie met non blocking fouten?"	Hugo	
21	18-9-2006	Bevindingenlijst: extra kolom met verwijzing naar charter(s): performance bevindingen verwijzen naar de performance charter.	Hugo	besluit
22	19-9-2006	Voorafgaand aan het testen van een charter door Jaap, kort overleg met Carin over blocking issue's voor de test	Jaap, Carin	besluit
23	19-9-2006	Carin geeft aan welke issue's blocking zijn zodat Jaap en Jasper naar Leo kunnen gaan om deze in de ACC te laten zetten	Carin	besluit, vervalt wordt vervangen door 25
24	19-9-2006	Bevindingenlijst: extra kolom met ernst, in welke charter geconstateerd is(met bevinding nummer) en in welke charter het getest zou moeten worden (=betrekking heeft)	Allen	besluit
25	19-9-2006	ipv besluit 23: Leo zet alle opgeloste issue (na test in de DVL) in de ACC, met gelijktijdig mail richting jaap en carin	Leo, hugo	besluit
26	28-9-2006	Charter 16 wordt geschrapt, omdat het voldoende getest is in de andere charters	Jaap Matthijs Jasper Derk-Jan	besluit

De-briefing sessions

- Status overview of the carters
- Charters in print, or beamer
- Stakeholders – group decisions



Collis
www.collis.nl

17

The ideal test-session length

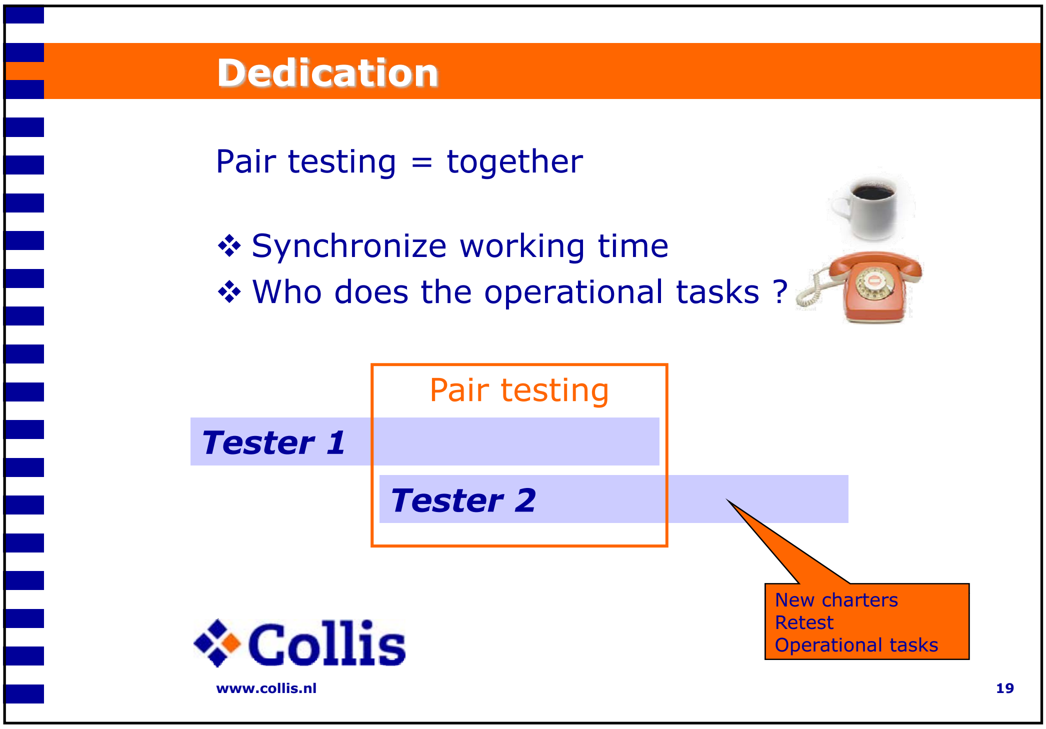
- Session length: Workshops ET held at Collis learned that:

  ```
  On 4 hour session take
  30 minutes preparation
  3 hour test execution + finding registration
  15 minutes evaluation
  ```

- One debriefing session a day
  (but start with 2 sessions)

Collis
www.collis.nl

18



Dedication

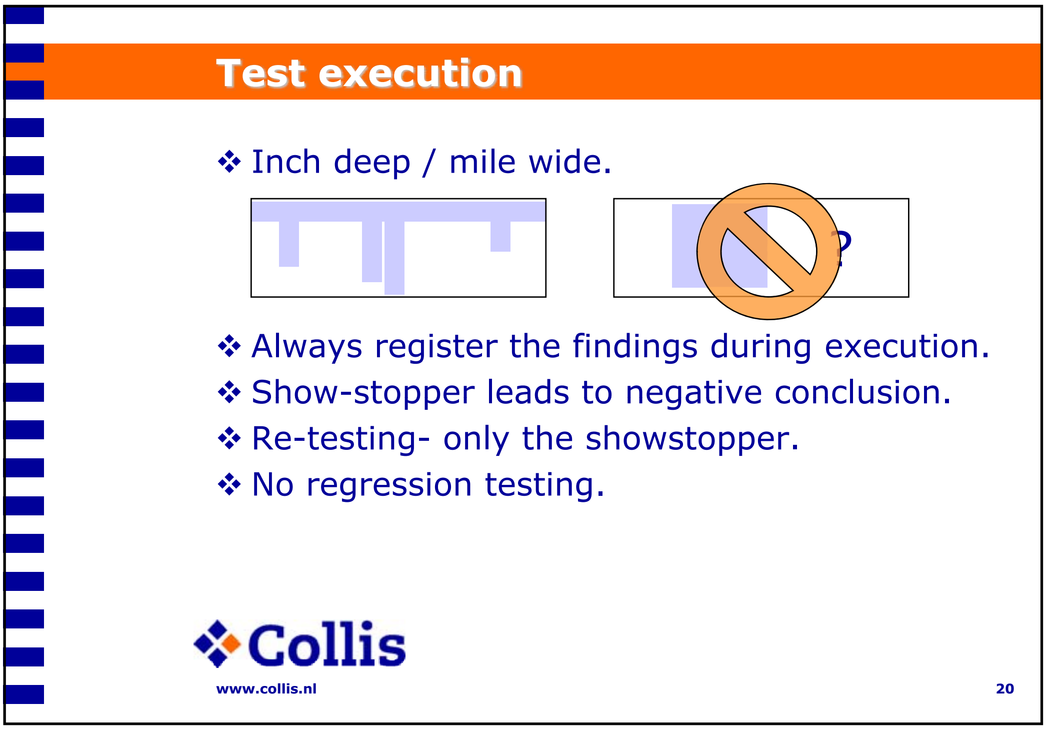
Pair testing = together

❖ Synchronize working time
❖ Who does the operational tasks ?

Pair testing
Tester 1
Tester 2
New charters
Retest
Operational tasks

Collis
www.collis.nl

19



Test execution

❖ Inch deep / mile wide.

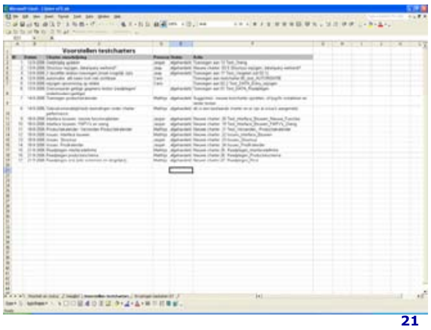
❖ Always register the findings during execution.
❖ Show-stopper leads to negative conclusion.
❖ Re-testing- only the showstopper.
❖ No regression testing.

Collis
www.collis.nl

20

Adding and adapting test charters

- ❖ Original based on functional breakdown
- ❖ Added charters for
  - ❖ Scenario’s
  - ❖ Changes affecting other functions
  - ❖ Multi user/Authorisation/User friendliness
  - ❖ Error guessing (last day)



Voorstellen testcharters

Collis

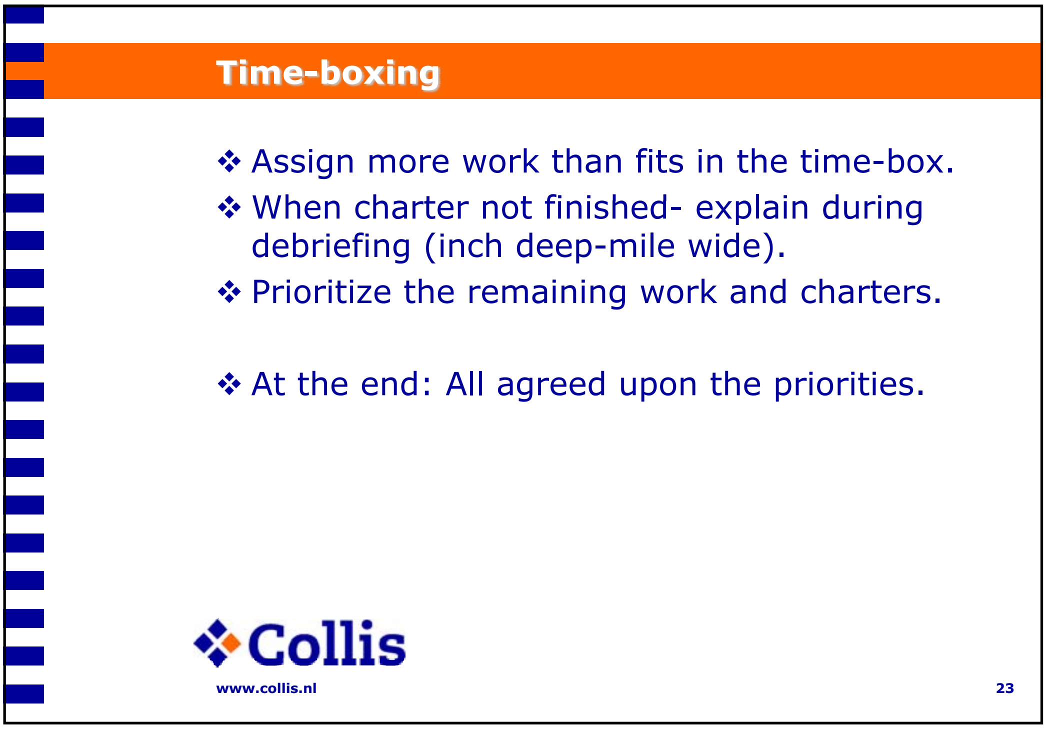
www.collis.nl

Testlog

- ❖ Tests are logged in the charter
- ❖ Used for regression testing

Collis

www.collis.nl



Time-boxing

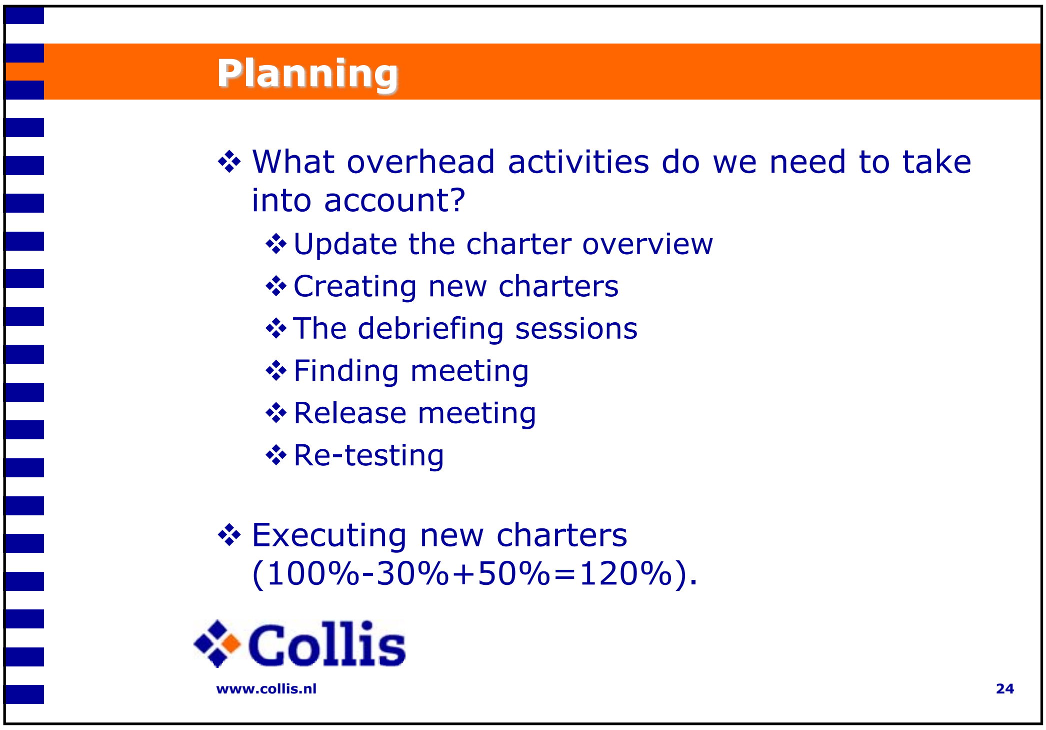
- Assign more work than fits in the time-box.
- When charter not finished- explain during debriefing (inch deep-mile wide).
- Prioritize the remaining work and charters.

- At the end: All agreed upon the priorities.

Collis

www.collis.nl

23



Planning

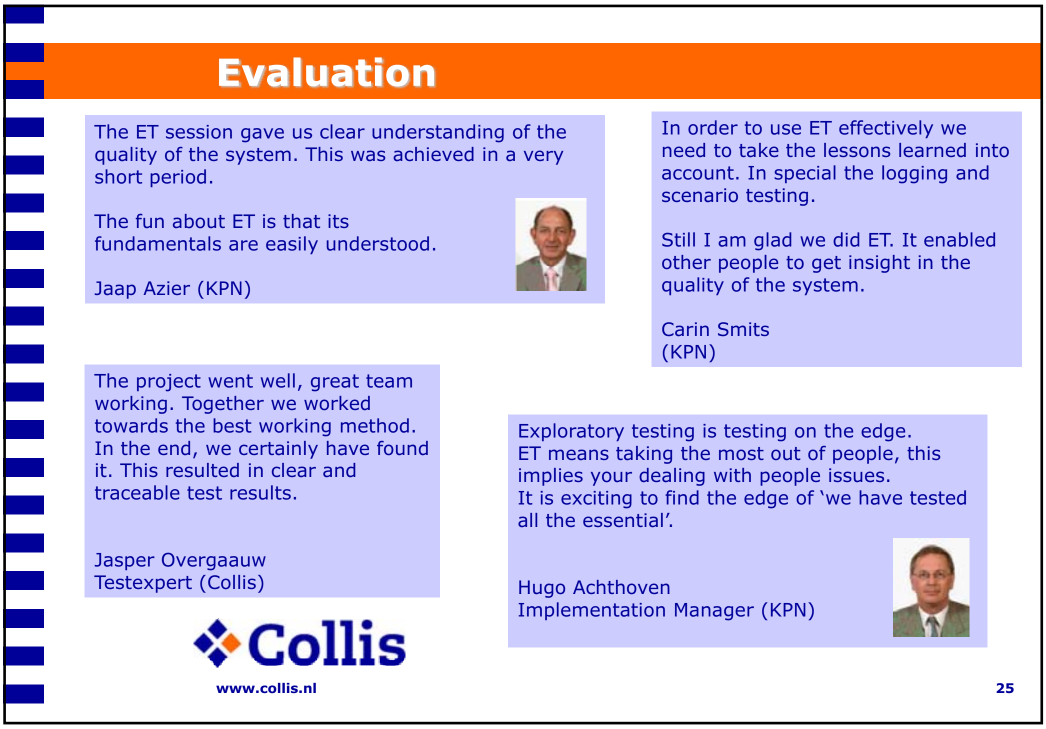
- What overhead activities do we need to take into account?
  - Update the charter overview
  - Creating new charters
  - The debriefing sessions
  - Finding meeting
  - Release meeting
  - Re-testing

- Executing new charters (100%-30%+50%=120%).

Collis

www.collis.nl

24



Evaluation

The ET session gave us clear understanding of the quality of the system. This was achieved in a very short period.

The fun about ET is that its fundamentals are easily understood.

Jaap Azier (KPN)

In order to use ET effectively we need to take the lessons learned into account. In special the logging and scenario testing.

Still I am glad we did ET. It enabled other people to get insight in the quality of the system.

Carin Smits
(KPN)

The project went well, great team working. Together we worked towards the best working method. In the end, we certainly have found it. This resulted in clear and traceable test results.

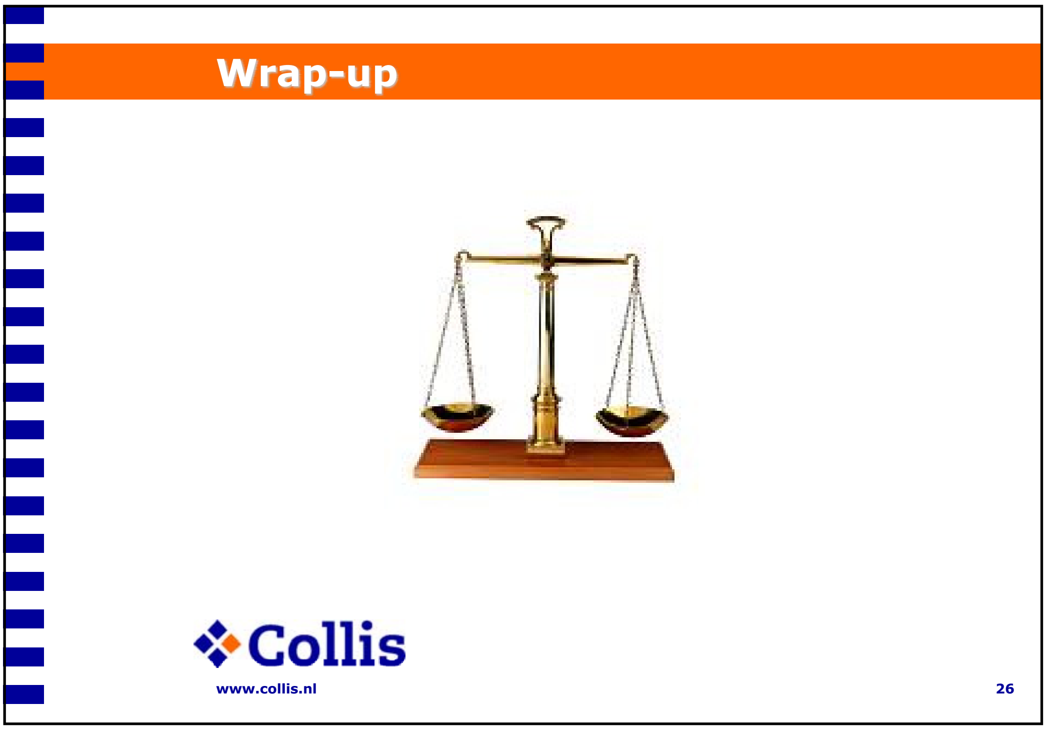
Jasper Overgaauw
Testexpert (Collis)

Exploratory testing is testing on the edge.
ET means taking the most out of people, this implies your dealing with people issues.
It is exciting to find the edge of 'we have tested all the essential'.

Hugo Achthoven
Implementation Manager (KPN)

Collis
www.collis.nl

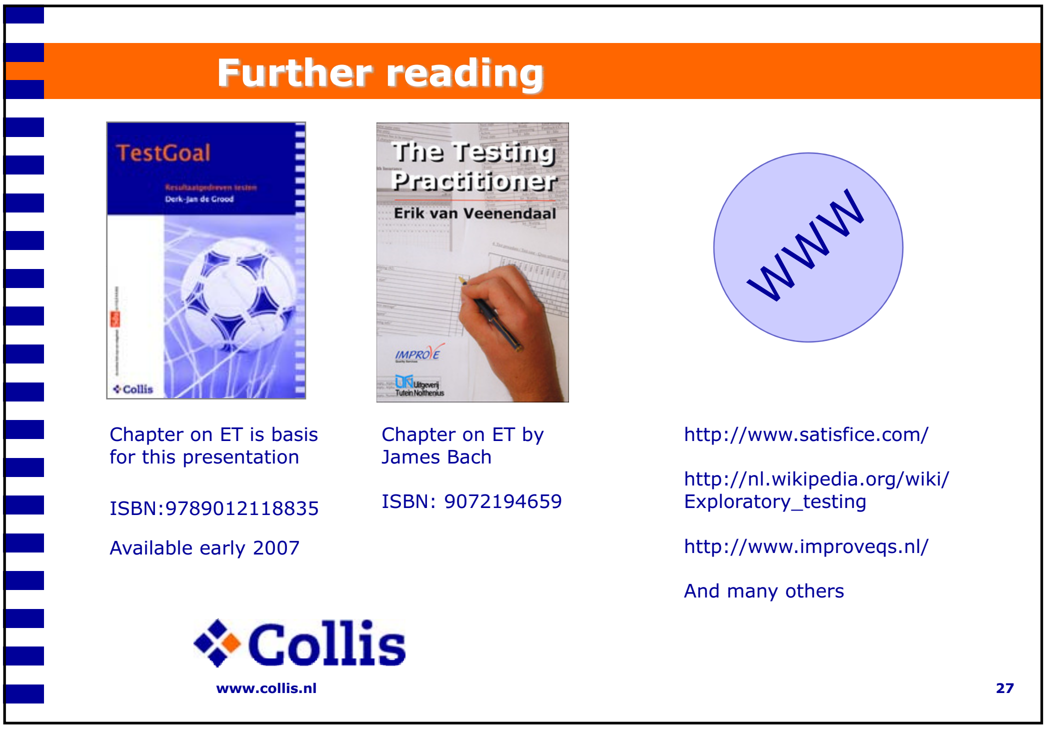
25



Wrap-up

Collis
www.collis.nl

26



Further reading

Chapter on ET is basis for this presentation

ISBN:9789012118835

Available early 2007

Chapter on ET by James Bach

ISBN: 9072194659

http://www.satisfice.com/

http://nl.wikipedia.org/wiki/Exploratory_testing

http://www.improveqs.nl/

And many others

Collis
www.collis.nl

27

Questions ?

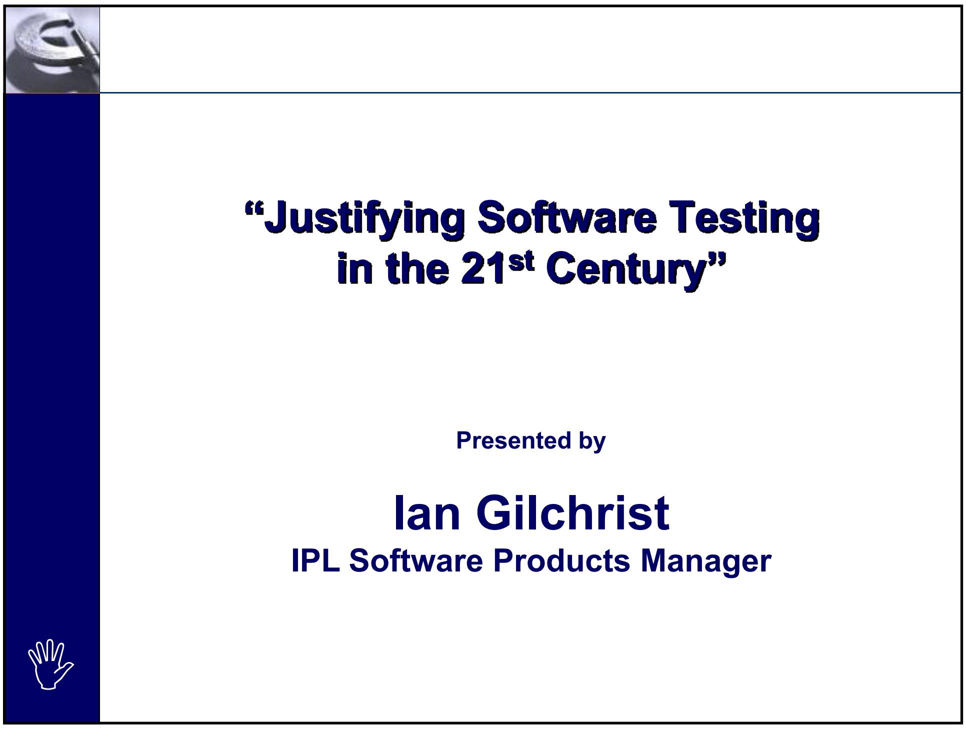
Derk-Jan de Grood
grood@Collis.nl

Collis
De Heijderweg 1
2314 XZ Leiden
The Netherlands

Acknowledgements:
Jaap Azier – KPN Telecom
Jasper Overgouw - Collis

Collis
www.collis.nl

28

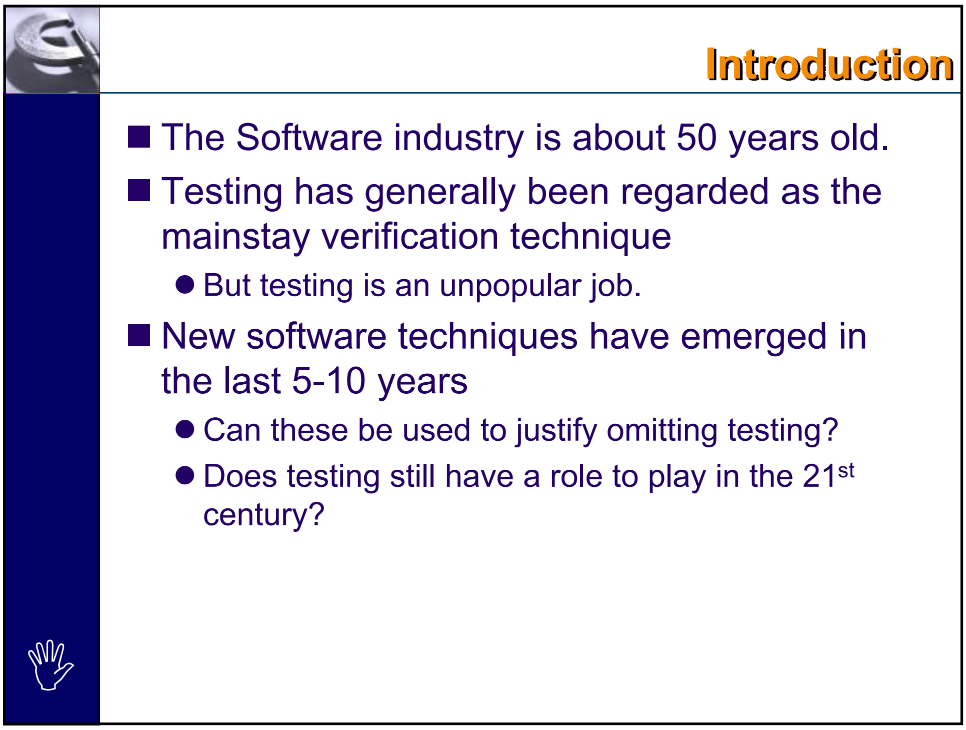


"Justifying Software Testing in the 21st Century"

Presented by

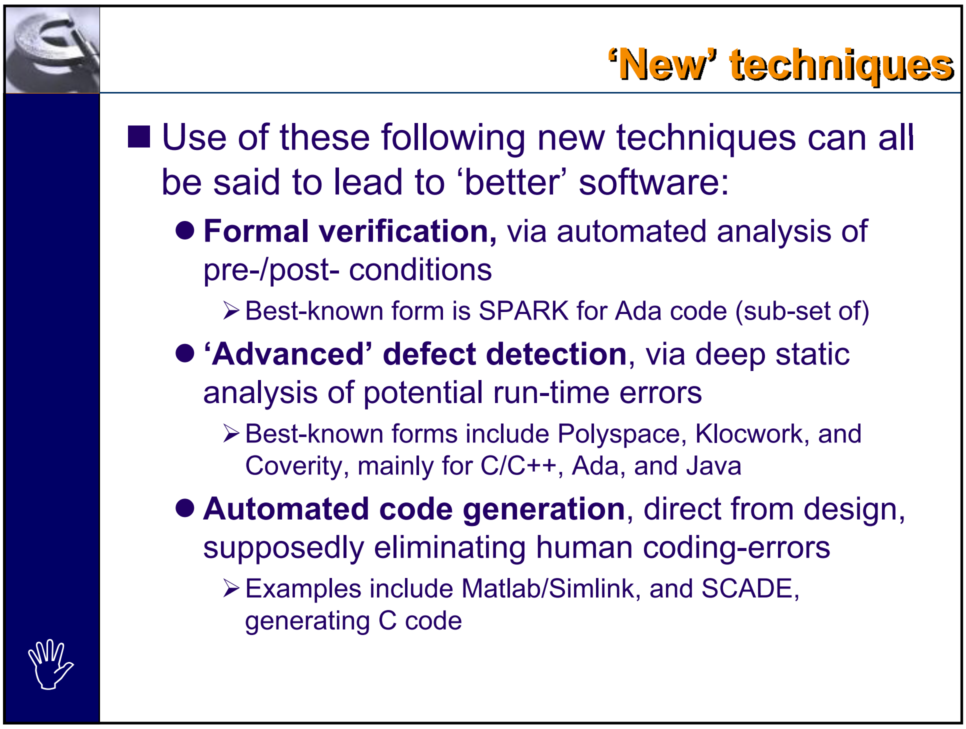
Ian Gilchrist

IPL Software Products Manager



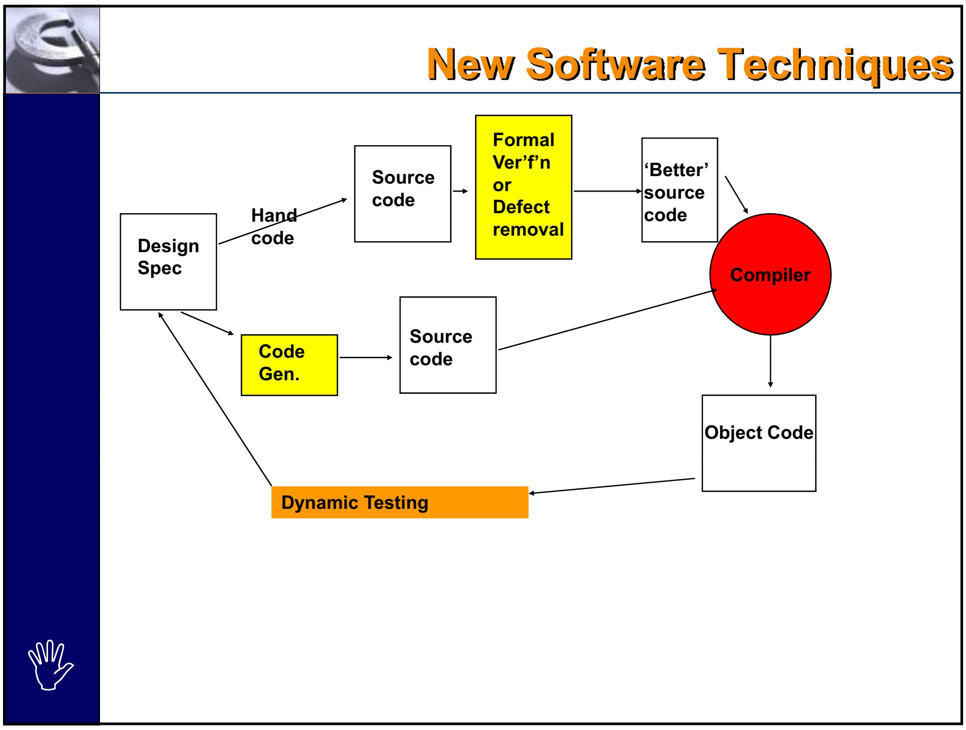
Introduction

- The Software industry is about 50 years old.
- Testing has generally been regarded as the mainstay verification technique
  - But testing is an unpopular job.
- New software techniques have emerged in the last 5-10 years
  - Can these be used to justify omitting testing?
  - Does testing still have a role to play in the 21st century?

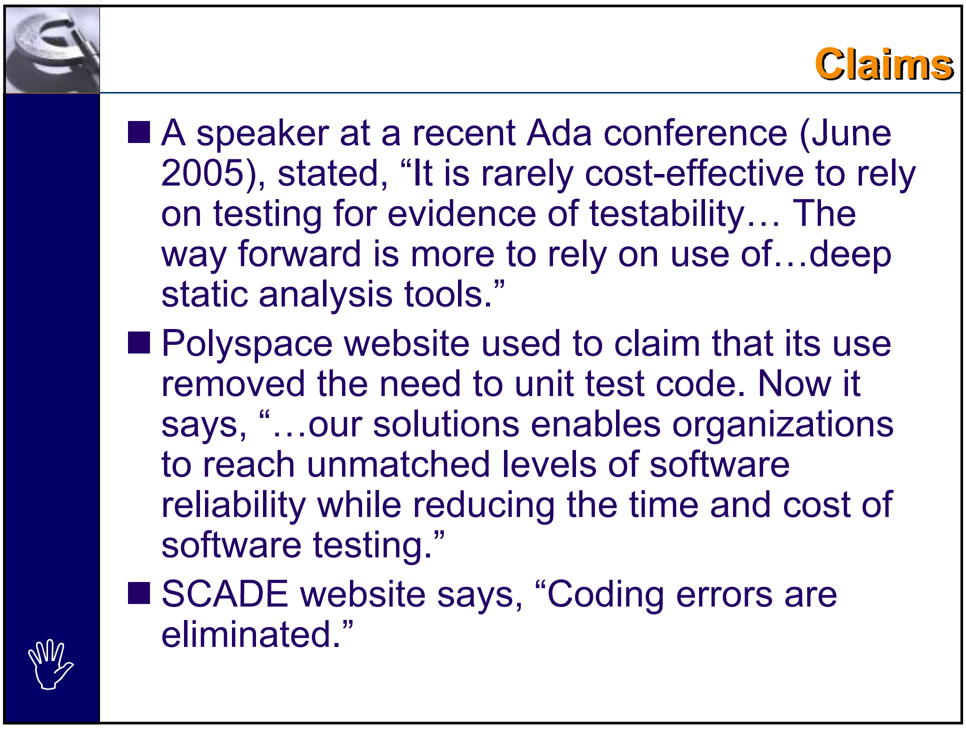


‘New’ techniques

- Use of these following new techniques can all be said to lead to ‘better’ software:
  - **Formal verification,** via automated analysis of pre-/post- conditions
    - Best-known form is SPARK for Ada code (sub-set of)
  - **‘Advanced’ defect detection**, via deep static analysis of potential run-time errors
    - Best-known forms include Polyspace, Klocwork, and Coverity, mainly for C/C++, Ada, and Java
  - **Automated code generation**, direct from design, supposedly eliminating human coding-errors
    - Examples include Matlab/Simlink, and SCADE, generating C code

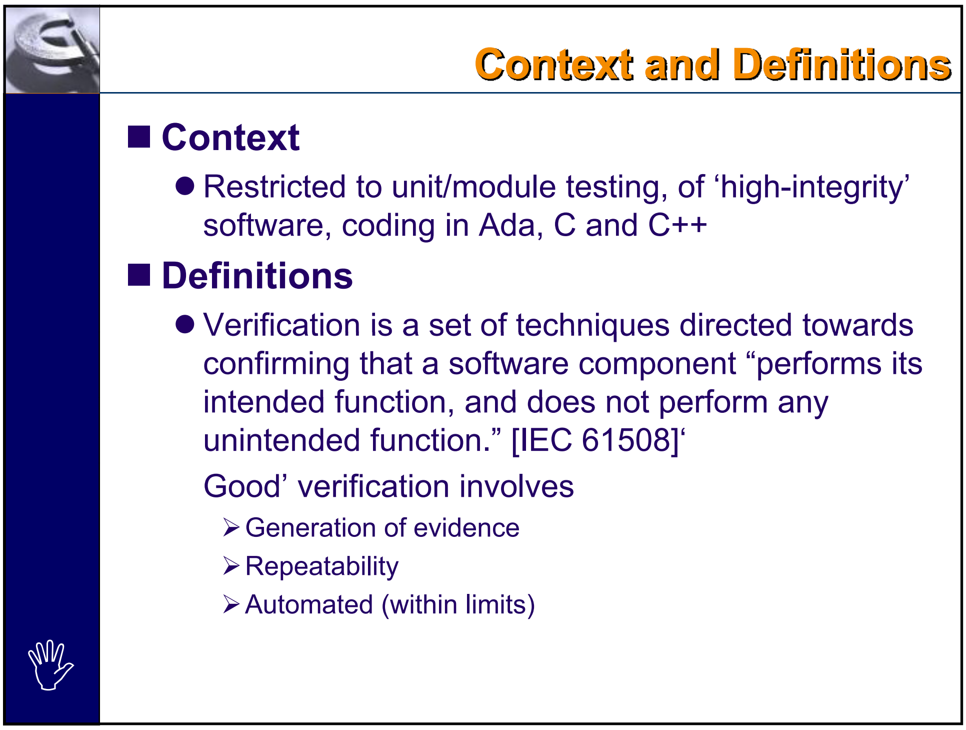


New Software Techniques



Claims

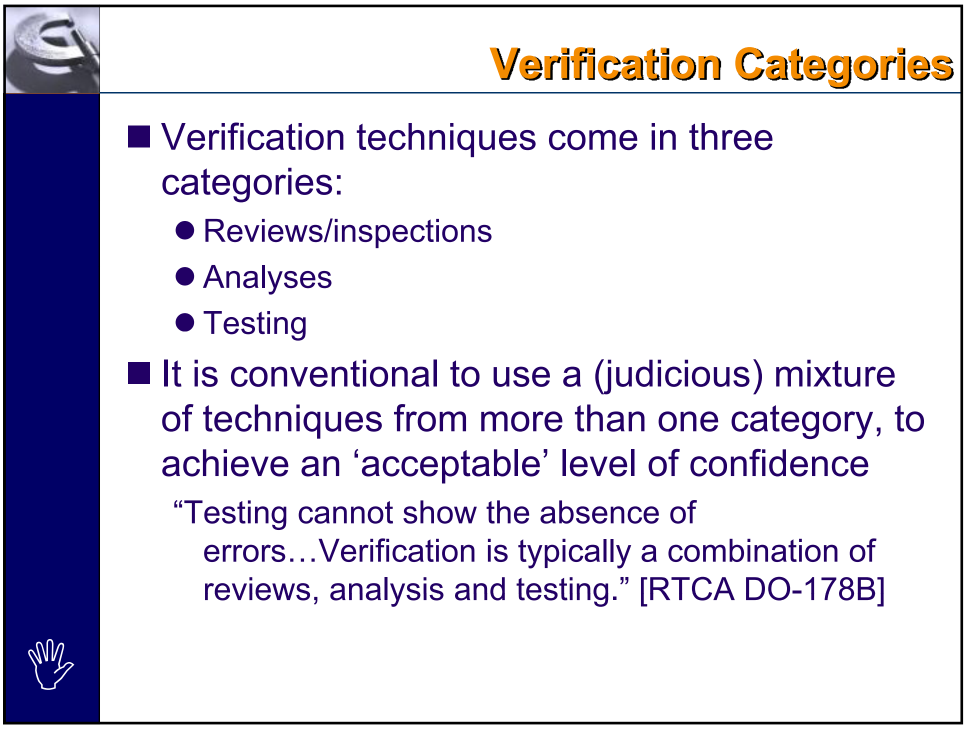
- A speaker at a recent Ada conference (June 2005), stated, “It is rarely cost-effective to rely on testing for evidence of testability… The way forward is more to rely on use of…deep static analysis tools.”
- Polyspace website used to claim that its use removed the need to unit test code. Now it says, “…our solutions enables organizations to reach unmatched levels of software reliability while reducing the time and cost of software testing.”
- SCADE website says, “Coding errors are eliminated.”



Context and Definitions

- **Context**
  - Restricted to unit/module testing, of ‘high-integrity’ software, coding in Ada, C and C++
- **Definitions**
  - Verification is a set of techniques directed towards confirming that a software component “performs its intended function, and does not perform any unintended function.” [IEC 61508]‘

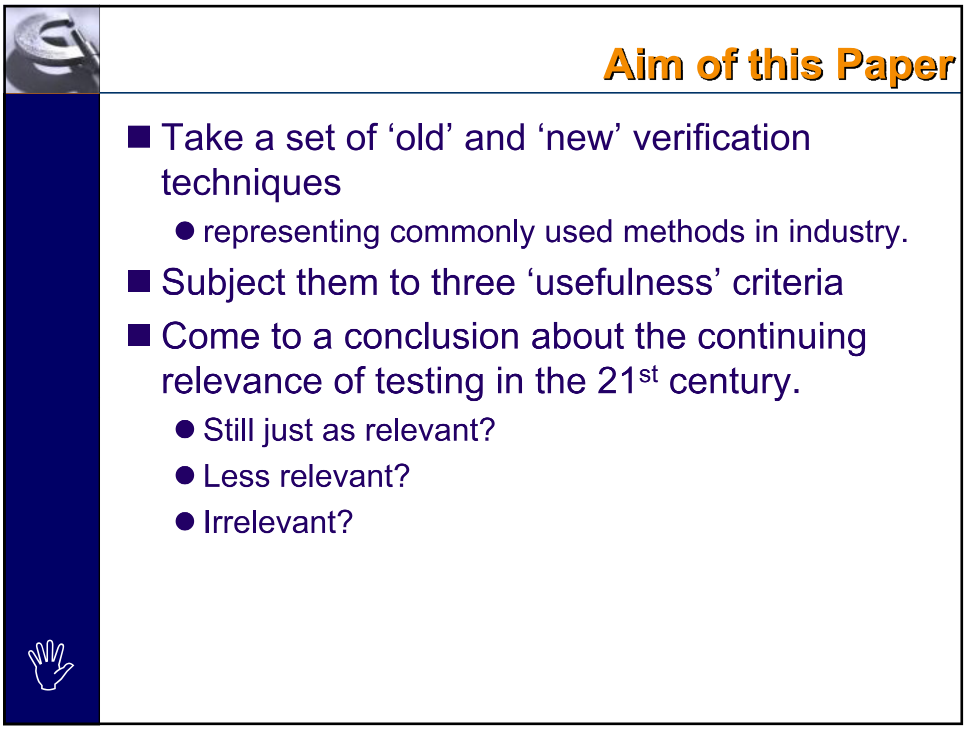
    Good’ verification involves
    - Generation of evidence
    - Repeatability
    - Automated (within limits)



Verification Categories

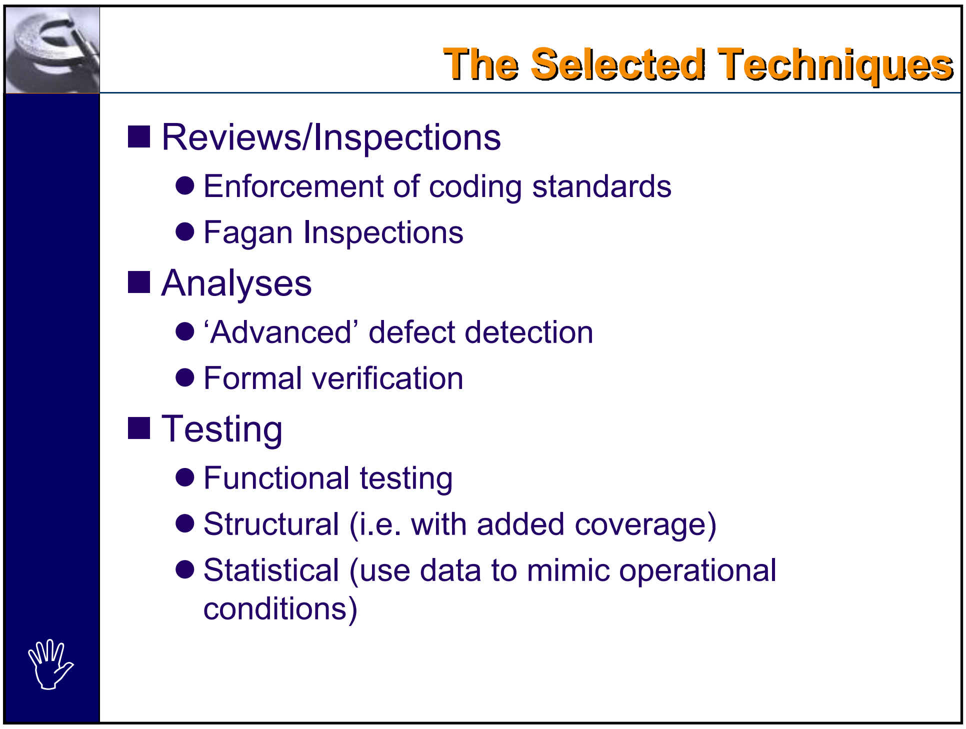
- Verification techniques come in three categories:
  - Reviews/inspections
  - Analyses
  - Testing
- It is conventional to use a (judicious) mixture of techniques from more than one category, to achieve an 'acceptable' level of confidence

  "Testing cannot show the absence of errors…Verification is typically a combination of reviews, analysis and testing." [RTCA DO-178B]



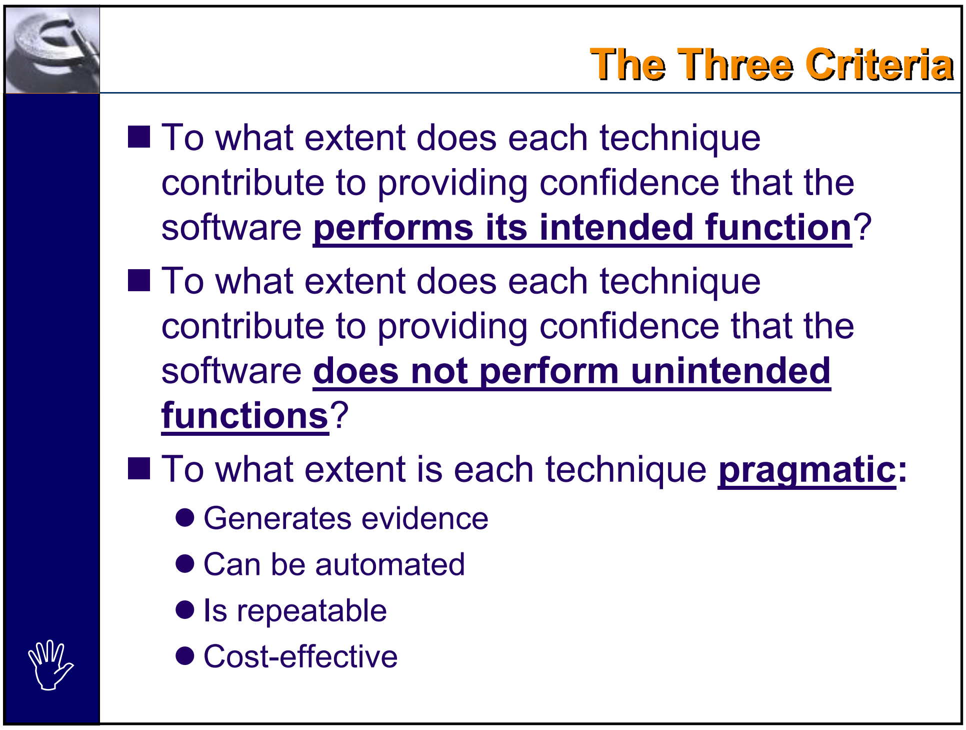
Aim of this Paper

- Take a set of 'old' and 'new' verification techniques
  - representing commonly used methods in industry.
- Subject them to three 'usefulness' criteria
- Come to a conclusion about the continuing relevance of testing in the 21st century.
  - Still just as relevant?
  - Less relevant?
  - Irrelevant?



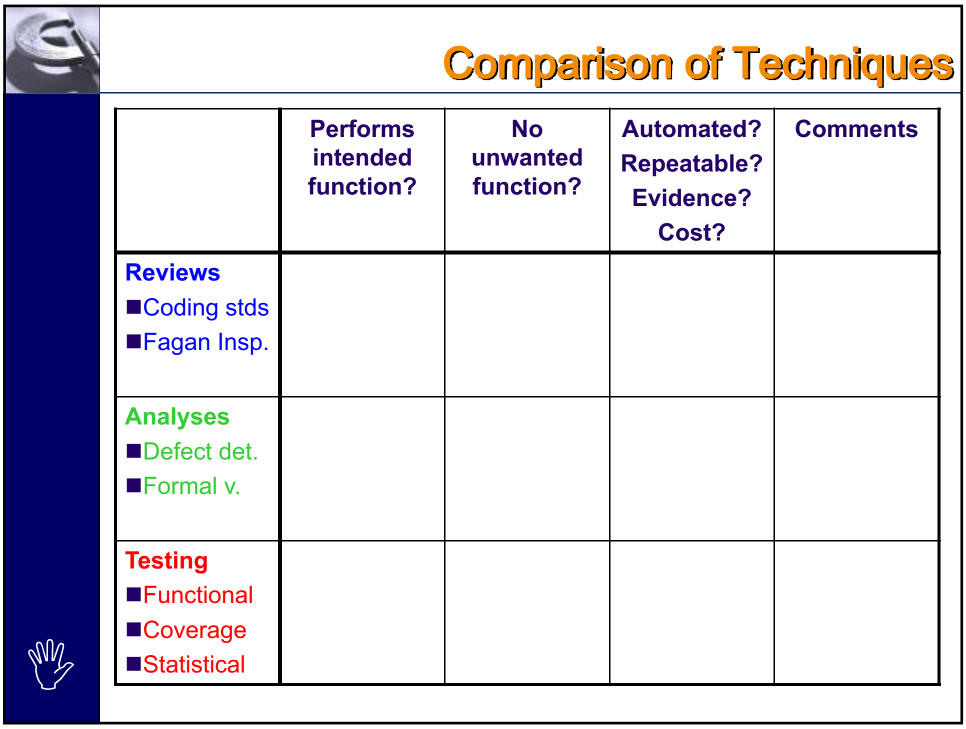
The Selected Techniques

- Reviews/Inspections
  - Enforcement of coding standards
  - Fagan Inspections
- Analyses
  - ‘Advanced’ defect detection
  - Formal verification
- Testing
  - Functional testing
  - Structural (i.e. with added coverage)
  - Statistical (use data to mimic operational conditions)



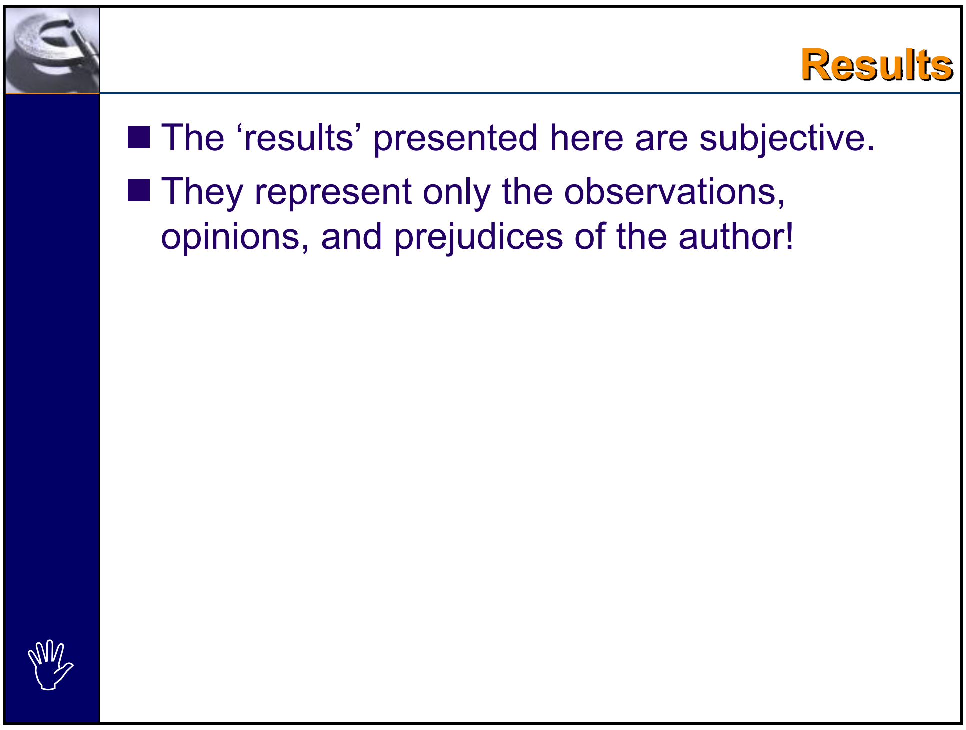
The Three Criteria

- To what extent does each technique contribute to providing confidence that the software **performs its intended function**?
- To what extent does each technique contribute to providing confidence that the software **does not perform unintended functions**?
- To what extent is each technique **pragmatic:**
  - Generates evidence
  - Can be automated
  - Is repeatable
  - Cost-effective



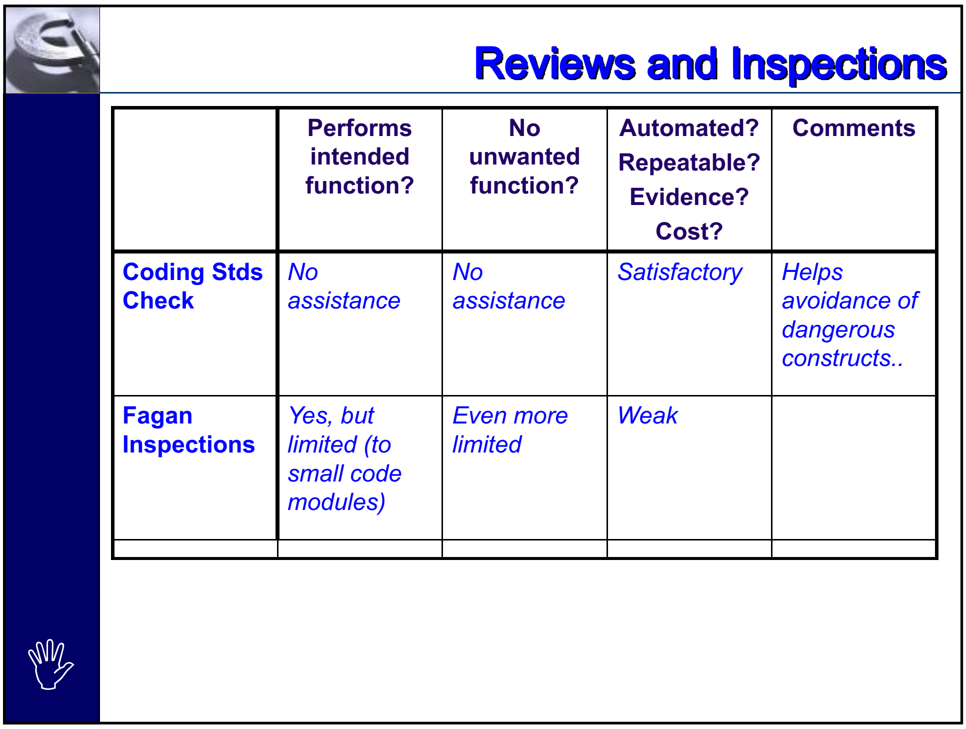
Comparison of Techniques

	Performs intended function?	No unwanted function?	Automated? Repeatable? Evidence? Cost?	Comments
Reviews ■Coding stds ■Fagan Insp.				
Analyses ■Defect det. ■Formal v.				
Testing ■Functional ■Coverage ■Statistical				



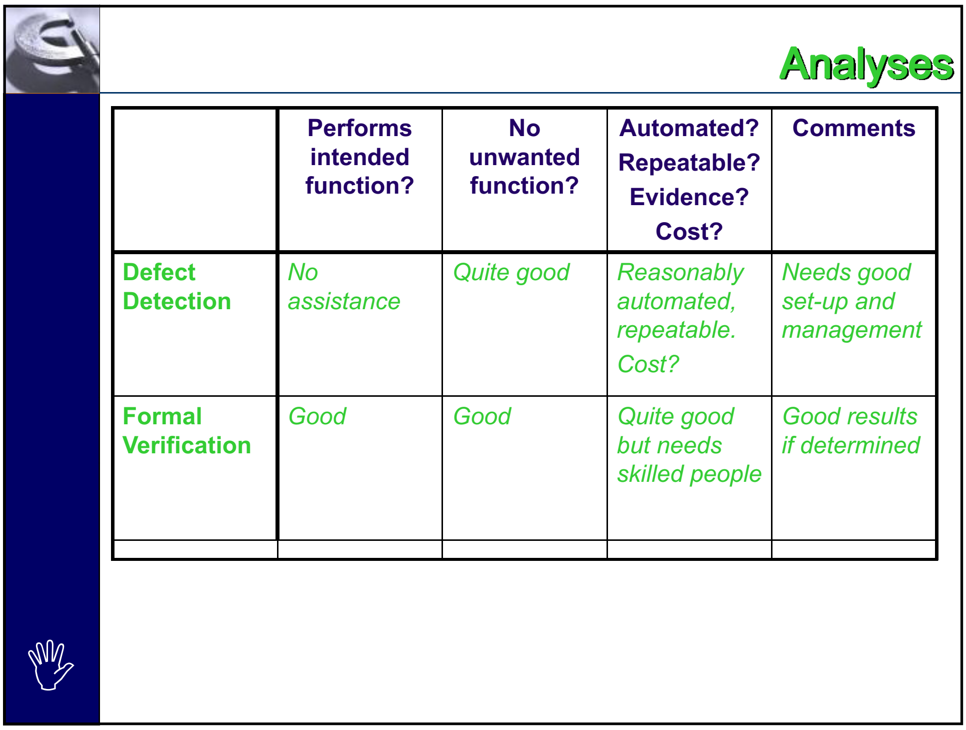
Results

- The ‘results’ presented here are subjective.
- They represent only the observations, opinions, and prejudices of the author!



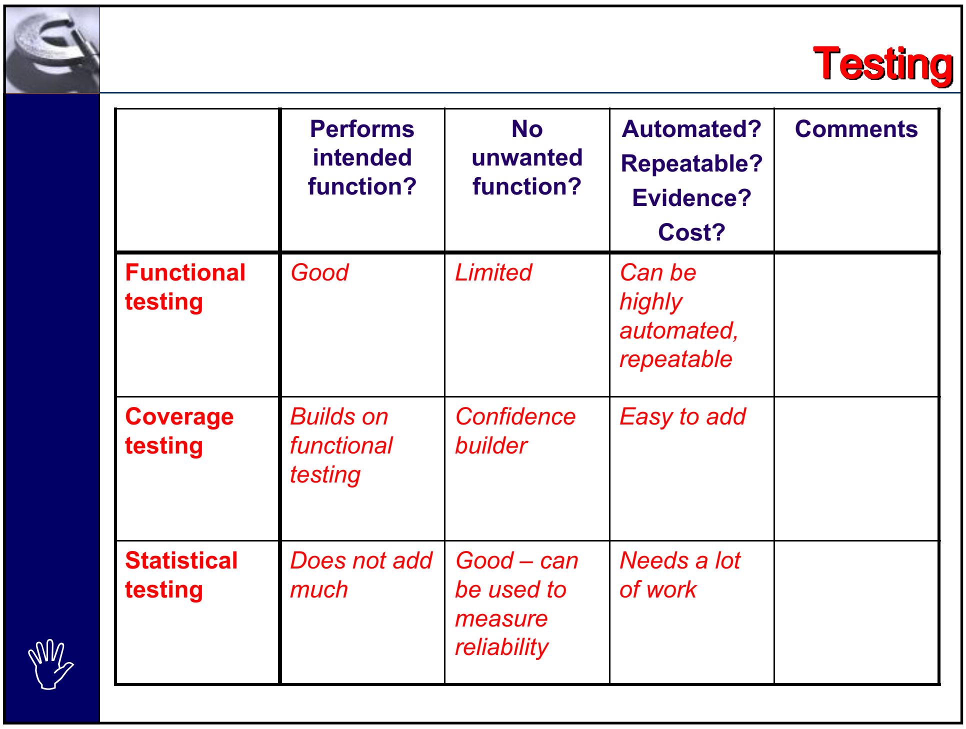
Reviews and Inspections

	Performs intended function?	No unwanted function?	Automated? Repeatable? Evidence? Cost?	Comments
Coding Stds Check	*No assistance*	*No assistance*	*Satisfactory*	*Helps avoidance of dangerous constructs..*
Fagan Inspections	*Yes, but limited (to small code modules)*	*Even more limited*	*Weak*	



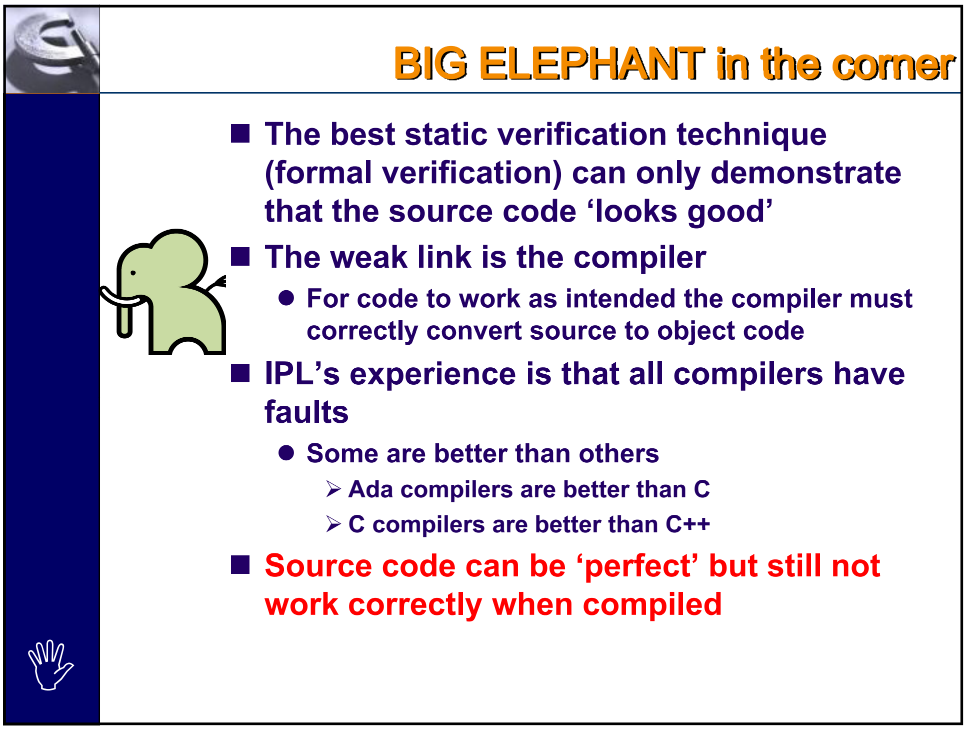
Analyses

	Performs intended function?	No unwanted function?	Automated? Repeatable? Evidence? Cost?	Comments
Defect Detection	*No assistance*	*Quite good*	*Reasonably automated, repeatable. Cost?*	*Needs good set-up and management*
Formal Verification	*Good*	*Good*	*Quite good but needs skilled people*	*Good results if determined*



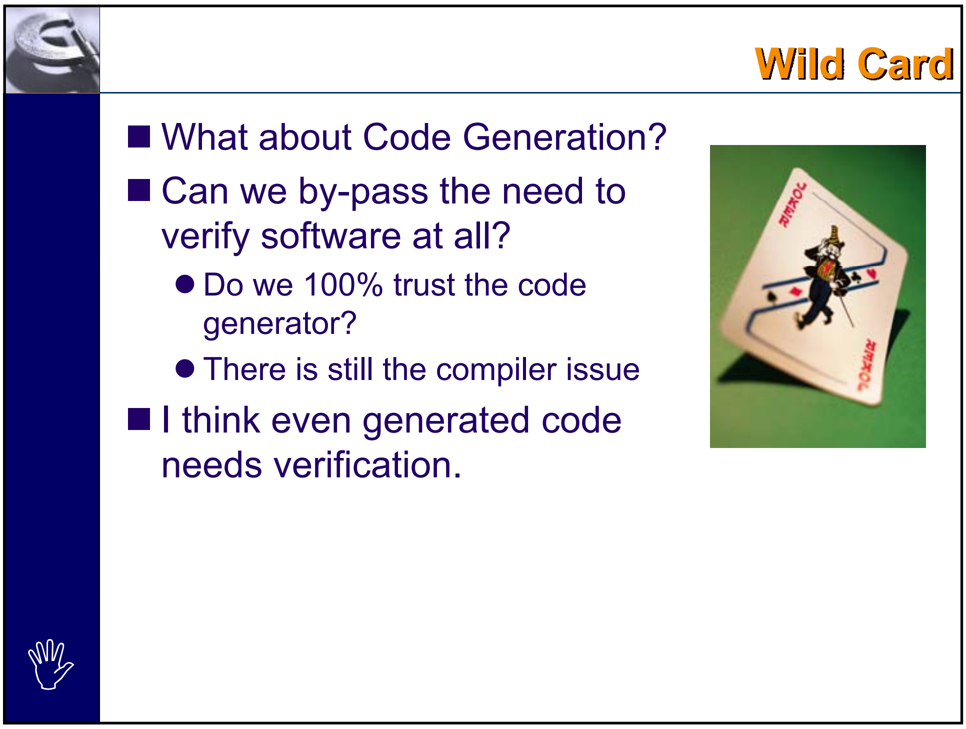
Testing

	Performs intended function?	No unwanted function?	Automated? Repeatable? Evidence? Cost?	Comments
Functional testing	*Good*	*Limited*	*Can be highly automated, repeatable*	
Coverage testing	*Builds on functional testing*	*Confidence builder*	*Easy to add*	
Statistical testing	*Does not add much*	*Good – can be used to measure reliability*	*Needs a lot of work*	



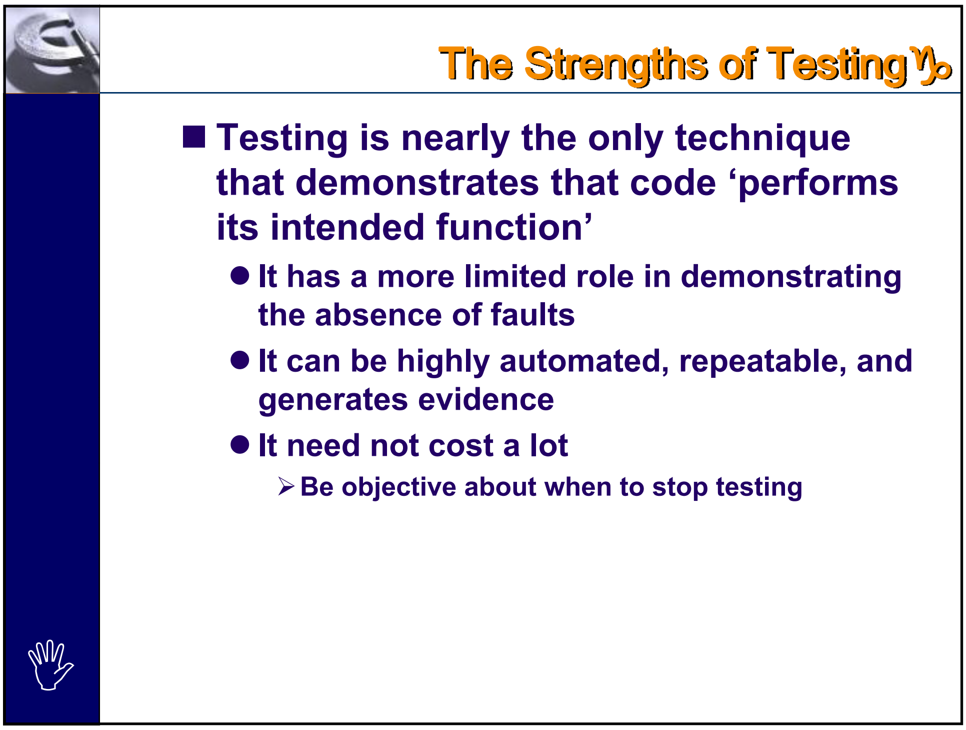
BIG ELEPHANT in the corner

- The best static verification technique (formal verification) can only demonstrate that the source code 'looks good'
- The weak link is the compiler
  - For code to work as intended the compiler must correctly convert source to object code
- IPL's experience is that all compilers have faults
  - Some are better than others
    - Ada compilers are better than C
    - C compilers are better than C++
- Source code can be 'perfect' but still not work correctly when compiled



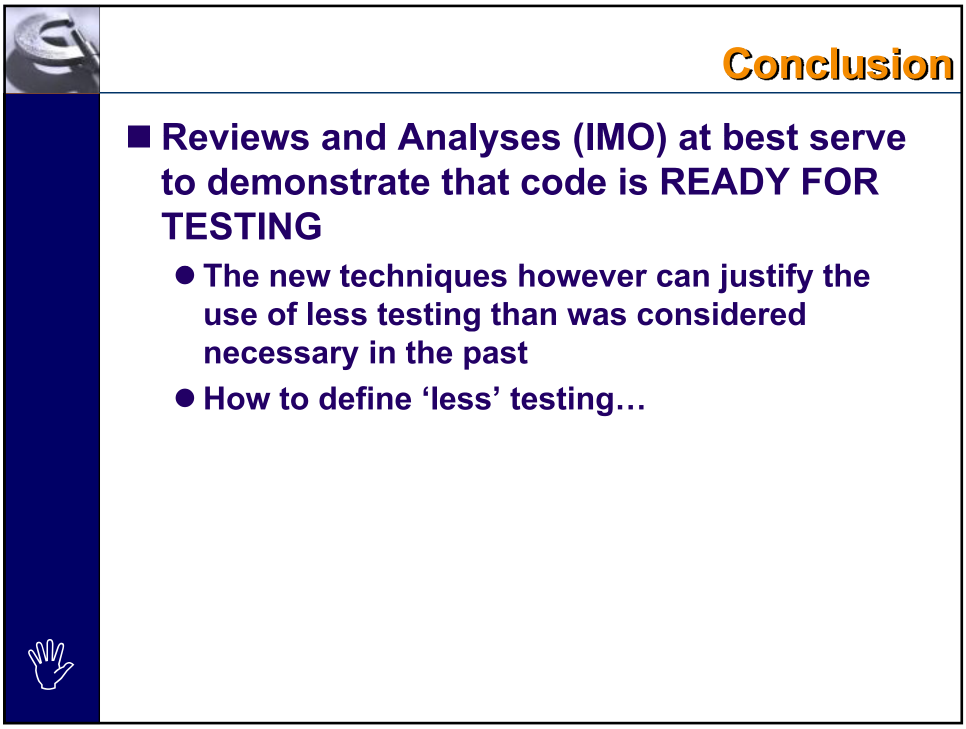
Wild Card

- What about Code Generation?
- Can we by-pass the need to verify software at all?
  - Do we 100% trust the code generator?
  - There is still the compiler issue
- I think even generated code needs verification.



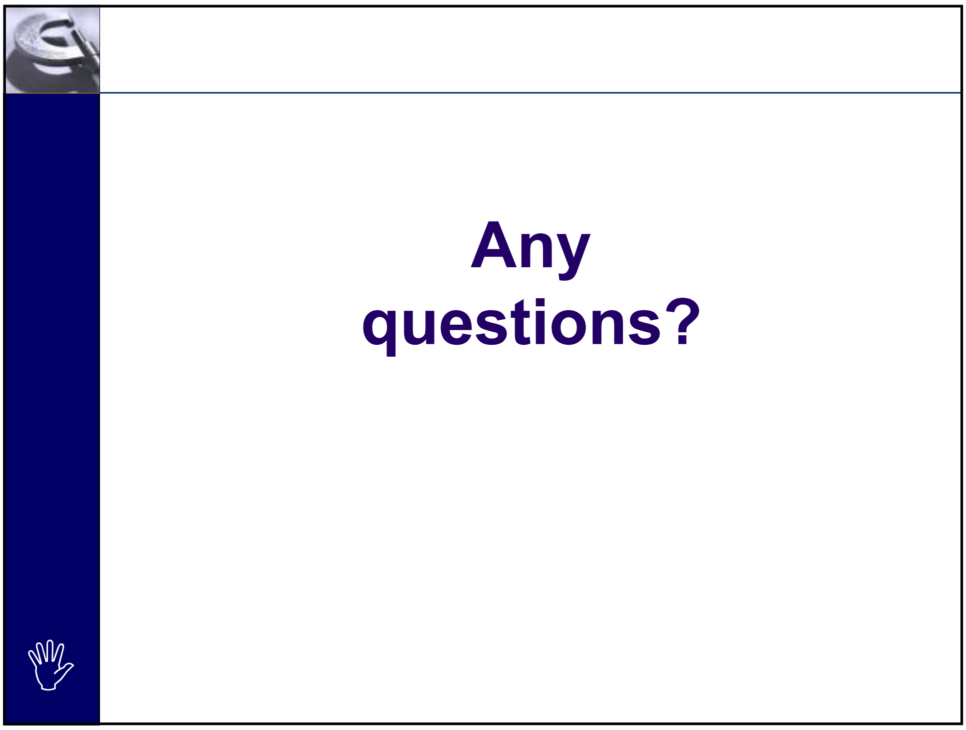
The Strengths of Testing ♑

- **Testing is nearly the only technique that demonstrates that code ‘performs its intended function’**
  - **It has a more limited role in demonstrating the absence of faults**
  - **It can be highly automated, repeatable, and generates evidence**
  - **It need not cost a lot**
    - **Be objective about when to stop testing**



Conclusion

- Reviews and Analyses (IMO) at best serve to demonstrate that code is READY FOR TESTING
  - The new techniques however can justify the use of less testing than was considered necessary in the past
  - How to define 'less' testing…



Any questions?

A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains

Hichem Boudali, Pepijn Crouzen, and Mariëlle Stoelinga

Department of Computer Science, University of Twente,
P.O.Box 217, 7500AE Enschede, The Netherlands.

Dynamic fault trees (DFTs) are a versatile and common formalism to model and analyze the failure of computer-based systems. This research consists in formalizing the semantics of DFTs in terms of input/output interactive Markov chains (I/O-IMCs), which extend continuous-time Markov chains with discrete input, output and internal actions.

Standard Fault trees (FTs) [2] provide an intuitive, high level, graphical formalism to model and analyze system failures. A FT model describes the system failure in terms of the failure of its components and is made up of basic events and logical gates, such as OR and AND gates. System unreliability (i.e. failure probability) is a typical performance measure of interest obtained from the analysis of the fault tree. Dynamic fault trees [4] extend standard (static) FTs by allowing the modeling of complex behaviors and interactions between components. More specifically they take into consideration not only the combination of failures but also the order in which they occur. A DFT is typically analyzed by first converting it into a Markov chain (MC) and then analyzing the MC. Due to the conversion into a MC, the DFT analysis suffers from the well-known state-space explosion problem.

As in standard fault trees, a DFT is constructed using a set of elements: basic events and gates. These elements and their interactions have been described informally in [4, 1]. Unfortunately, the lack of formality has led, in some cases, to undefined behavior and misinterpretations of the DFT, and more importantly to inconsistencies in the implementation of supporting tools such as Galileo [8].

In this work, we formalize the semantics of DFTs in terms of input-output interactive Markov chains (I/O-IMCs). IMCs [5] are an extension of continuous-time Markov chains with discrete actions and have proved to be a powerful and versatile formalism for a variety of applications. In particular, IMCs enable efficient algorithms for aggregating equivalent (i.e. weakly bisimilar) states. For our purposes, we needed IMCs that distinguish between input and output actions, and hence we combined IMCs with features from the I/O automaton model in [7]. Formalizing the semantics is important, since it provides a rigorous basis for analyzing DFTs and for developing and implementing tools that support DFTs. I/O-IMCs enable parallel composition and aggregation (i.e. state-space reduction), and also allow to model probabilistic as well as non-deterministic choices. The semantics we define is compositional. More specifically, we define the semantics of each DFT gate and basic event as an elementary I/O-IMC. The semantics of the whole DFT is obtained by parallel composing the I/O-IMCs of the various gates and basic events contained in the DFT. We believe that our approach, compared to earlier informal and formal DFT semantics, has several important advantages: it is based on a formal Markovian process algebra, it is compositional, easy to understand and implement, allows for non-determinism, it alleviates the state-space explosion problem, and provides a readily extensible framework (by adding new types of DFT elements and defining their corresponding I/O-IMCs). We use a *compositional aggregation* approach to build the I/O-IMC of the whole DFT. A DFT is essentially, as described above, a collection of elementary I/O-IMCs. In order to build the I/O-IMC of the DFT, we (1) parallel compose two I/O-IMCs, and then (2) aggregate equivalent (i.e. weakly bisimilar) states (thus effectively reducing the state space). We keep repeating and interleaving these two steps until we are left with a single aggregated I/O-IMC. This compositional aggregation approach is crucial in alleviating the state-space explosion problem.

Being a first step, we have restricted our attention to DFT components with exponential failure rates, but the same approach could be taken for other probability distributions. Since IMCs only support exponential distributions, this would require the use for instance of phase-type distributions, or the use of a different underlying formalism other than IMCs, e.g. stochastic automata [3].

We have implemented our approach in a prototype tool that takes as input a DFT in the format of the Galileo tool and produces as output a collection of I/O-IMCs in the format of the TIPP tool [6]. The TIPP tool is then called through a number of iterations to perform the compositional aggregation steps and to finally compute the system failure probability. We have applied this theory to several case studies which illustrate the feasibility of our approach and its effectiveness in reducing the state-space to be analysed.

References

1. M A Boyd. *Dynamic fault tree models: techniques for analyses of advanced fault tolerant computer systems.* PhD dissertation, Dept. of Computer Science, Duke University, 1991.
2. United States Nuclear Regulatory Commission. Fault tree handbook, NUREG-0492. Technical report, NASA, 1981.
3. P R D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* PhD thesis, Department of Computer Science, University of Twente, Nov 1999.
4. J B Dugan, S J Bavuso, and M A Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–377, September 1992.
5. H Hermanns. *Interactive Markov Chains*, volume 2428/2002 of *Lecture Notes in Computer Science.*
6. H Hermanns, U Herzog, U Klehmet, V Mertsiotakis, and M Siegle. Compositional performance modelling with the TIPPtool. *Lecture Notes in Computer Science*, 1469:51–62, 1998.
7. N A Lynch and M R Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.
8. K J Sullivan, J B Dugan, and D Coppit. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232–235, June 1999.

Discovering Faults in Idiom-Based Exception Handling

Magiel Bruntink[*]
Magiel.Bruntink@cwi.nl

Arie van Deursen[*] [†]
Arie.van.Deursen@cwi.nl

Tom Tourwé[*] [‡]
Tom.Tourwe@cwi.nl

ABSTRACT

In this paper, we analyse the exception handling mechanism of a state-of-the-art industrial embedded software system. Like many systems implemented in classic programming languages, our subject system uses the popular return-code idiom for dealing with exceptions. Our goal is to evaluate the fault-proneness of this idiom, and we therefore present a characterisation of the idiom, a fault model accompanied by an analysis tool, and empirical data. Our findings show that the idiom is indeed fault prone, but that a simple solution can lead to significant improvements.

1. INTRODUCTION

A key component of any reliable software system is its exception handling. This allows the system to detect errors, and react to them correspondingly, for example by recovering the error or by signalling an appropriate error message. As such, exception handling is not an optional add-on, but a sine qua non: a system without proper exception handling is likely to crash continuously, which renders it useless for practical purposes.

Despite its importance, several studies have shown that exception handling is often the least well understood, documented and tested part of a system. For example, [30] states that more than 50% of all system failures in a telephone switching application are due to faults in exception handling algorithms, and [21] explains that the Ariane 5 launch vehicle was lost due to an unhandled exception.

Various explanations for this phenomenon have been given.

First of all, since exception handling is not the primary *concern* to be implemented, it does not receive as much attention in requirements, design and testing. [27] explains that exception handling design degrades (in part) because less attention is paid to it, while [9] explains that testing is often most thorough for the ordinary application functionality, and least thorough for the exception handling functionality. Granted, exception handling behaviour is hard to test, as the root causes that invoke the exception handling mechanism are often difficult to generate, and a combinatorial explosion of test cases is to be expected. Moreover, it is very hard to prepare a system for all possible errors that might occur at runtime. The environment in which the system will run is often unpredictable, and errors may thus occur for which a system did not prepare.

Second, exception handling functionality is crosscutting in the meanest sense of the word. [22] shows that even the simplest exception handling strategy takes up 11% of an application's implementation, that it is scattered over many different files and functions and that it is tangled with the application's main functionality. This has a severe impact on understandability and maintainability of the code in general and the exception handling code in particular, and makes it hard to ensure correctness and consistency of the latter code.

Last, many software systems in use today are written in older languages, such as C or Cobol, that do not provide explicit support for exception handling. Such support makes exception handling design easier, by providing language constructs and accompanying static compiler checks. In the absence of such support, systems typically resort to systematic coding idioms for implementing exception handling, as advocated by the well-known *return code* technique, used in many C programs and operating systems. As shown in [4], such idioms are not scalable and compromise correctness.

In this paper, we focus on the exception handling mechanism of a real-time embedded system, that is over 15 years old, is developed using a state-of-the-art development process, and consists of over 10 million lines of C code. The system applies (a variant of) the return code idiom consistently throughout the implementation. The central question we seek to address is the following: "how can we reduce the number of implementation faults related to exception handling implemented by means of the return code idiom?". In order to answer this general question, a number of more specific questions needs to be answered first.

1. What kinds of faults can occur? Answering this question requires an in-depth analysis of the return code idiom, and a fault model that covers the possible faults to which the idiom can lead;

2. Which of these faults do actually occur in the code? A fault model only predicts which faults can occur, but does not say which faults actually occur in the code. By carefully analysing the subject system (automatically) an estimate of the probability of a particular fault can be given;

[*] Centrum voor Wiskunde en Informatica, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

[†] Software Evolution Research Laboratory (SWERL), Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), Delft University, Mekelweg 4, 2628 CD Delft, The Netherlands.

[‡] Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium.

ICSE 2006, Sjanghai, China. In Preparation.

3. What are the primary causes of these faults? The fault model explains *when* a fault occurs, but does not explicitly state *why* it occurs. Because we need to analyse the source code in detail for detecting faults, we can also study the causes of these faults, as we will see;

4. Can we eliminate these causes, and if so, how? Once we know why these faults occur and how often, we can come up with alternative solutions for implementing exception handling that help developers in avoiding such faults. An alternative solution is only a first step, (automated) migration can then follow.

We believe that answers to these questions are of interest to a broader audience then the original developers of our subject system. Any software system that is developed in a language without exception handling support will suffer the same problems, and guidelines for avoiding such problems are more than welcome. In this paper we offer experience, an analysis approach, tool support, empirical data, and alternative solutions to such projects.

```
int f(int a, int* b) {
   int r = OK;
   bool allocated = FALSE;
   r = mem_alloc(10, (int *)b);
   allocated = (r == OK);
   if((r == OK) && ((a < 0) || (a > 10))) {
      r = PARAM_ERROR;
      LOG(r,OK);
   }
   if(r == OK) {
      r = g(a);
      if(r != OK) {
         LOG(LINKED_ERROR,r);
         r = LINKED_ERROR;
      }
   }
   if(r == OK)
      r = h(b);
   if((r != OK) && allocated)
      mem_free(b);
   return r;
}
```

Figure 1: Exception handling idiom at ASML.

2. RELATED WORK

Fault (Bug) Finding Recently a lot of techniques and tools have been developed that aim at either static fault finding or program verification. However, we are not aware of fault finding approaches specifically targeting exception handling faults.

Fault finding and program verification have different goals. On the one hand, fault finding's main concern is finding as many (potentially) harmful faults as possible. Therefore fault finding techniques usually sacrifice formal soundness in order to gain performance and thus the ability to analyse larger systems. Specifically, Metal [13], PREfix [7], and ESC [16] follow this approach. We were inspired by many of the ideas underlying Metal for the implementation of our tool (see Section 5).

Model checking is also used as a basis for fault finding techniques. CMC [24] is a model checker that does not require the construction of a separate model for the system to be checked. Instead, the implementation (code) itself is checked directly, allowing for effective fault finding. In [32] the authors show how CMC can be used to find faults in file system implementations.

On the other hand, program verification is focused on proving specified properties of a system. For instance, MOPS [8] is capable of proving the absence of certain security vulnerabilities. More general approaches are SLAM [2] and ESP [10]. While ESP is burdened by the formal soundness requirement, it has nevertheless been used to analyse programs of up to 140 KLOC.

Idiom Checkers A number of general-purpose tools have been developed that can find basic coding errors [18, 25, 14]. These tools are however incapable of verifying domain-specific coding idioms, such as the return code idiom. More advanced tools [12, 29] are restricted to detecting higher-level design flaws but are not applicable at the implementation level.

In [4], we present an idiom checker for the parameter checking idiom, also in use at ASML. This idiom, although much simpler, resembles the exception handling idiom, and the verifier is based on similar techniques as presented in this paper.

Exception Handling Several proposals exist for extending the C language with an exception handling mechanism. [20, 26] and [31] all define exception handling macro's that mimic a Java/C++ exception-handling mechanism. Although slightly varying in syntax and semantics, these proposals are all based around an idiom using the C `setjmp/longjmp` facility.

Exceptional C [17] is a more drastic, and as such more powerful, extension of C with exception handling constructs as present in modern programming languages. It allows developers to declare and raise exceptions and define appropriate handlers. A function's signature should specify the exceptions that the function can raise, which allows the preprocessor to check correctness. Standard C code is generated as a result.

All these proposals differ from our proposal (Section 7) in that our proposal still uses the return-code idiom, but makes it more robust by hiding (some of) the implementation details. This makes migration of the old mechanism to the new one easier, an important concern considering ASML's 10 MLoC code base.

Robillard and Murphy describe an exception flow model and a corresponding tool in [28] and [27], to analyse exception handling in Java applications. They show how exception structure can degrade and present a technique based on software compartmenting to counter this phenomenon. Their work differs from ours in that they reason about the application-specific design of exception handling, whereas we focus on the (implementation of) the exception handling mechanism itself.

3. CHARACTERISING THE RETURN CODE IDIOM

The central question we seek to answer is how we can reduce the number of faults related to exception handling implemented by means of the return code idiom. To arrive at the answer, we first of all need a clear description of the workings of (the particular variant of) the return code idiom at ASML. We use an existing model for exception handling mechanisms (EHM) [19] to distinguish the different components of the idiom. This allows us to identify and focus on the most error-prone components in the next sections. Furthermore, expressing our problem in terms of this general EHM model makes it easier to apply our results to other systems using similar approaches.

3.1 Terminology

An exception at ASML is "any abnormal situation found by the equipment that hampers or could hamper the production". Exceptions are logged in an *event log*, that provides information on the machine history to different stakeholders (such as service engineers, quality assurance department, etc).

The EHM itself is based on two requirements:

1. a function that detects an error should log that error in the event log, and recover it or pass it on to its caller;

2. a function that receives an error from a called function must provide useful context information (if possible) by *linking* an error to the received error, and recover the error or pass it on to the calling function.

An error that is detected by a function is called a *root* error, while an error that is linked to an error received from a function is called a *linked* error.

If correctly implemented, the EHM produces a chain of related consecutive errors in the event log. This chain is commonly referred to as the *error link tree*, and resembles a stack trace as output by the Java virtual machine, for example.

Because ASML uses the C programming language, and C does not have explicit support for exception handling, each function in the ASML source code follows the *return code* idiom. Figure 1 shows an example of such a function. We will now discuss this approach in more detail.

3.2 Exception Representation

An exception representation *defines what an exception is and how it is represented.* At ASML, a *singular* representation is used, in the form of an *error variable* of type `int`. Line 2 in Figure 1 shows a typical example of such an error variable, that is initialised to the `OK` constant. This variable is used throughout the function to hold an *error value*, i.e., either `OK` or any other constant to signal an error. The variable can be assigned a constant, as in lines 7 and 14, or can be assigned the result of a function call, as in lines 4, 11 and 18. If the function does not recover from an error itself, the value of the error should be propagated through the caller by the `return` statement (line 23).

Note that multiple error variables are sometimes needed, when dealing with functions executing in parallel or when cleaning up resources (see later). Only one error value can be returned by a function, however, so special arrangements are necessary when using multiple error variables. Although this is important for the correct operation of the EHM, it is not the primary focus of this paper, so we will not discuss it here in detail.

3.3 Exception Raising

Exception raising is *the notification of an exception occurrence*. Different mechanisms exist, of which ASML uses the *explicit control-flow transfer* variant: if a root error is encountered, the error variable is assigned a constant (see lines 6 − 9), the function logs the error, stops executing its normal behaviour, and notifies its caller of the error.

Logging occurs by means of the `LOG` function (line 8), where the first argument is the new error encountered, which is linked to the second argument, that represents the previous error value. The function treats root errors as a special case of linked errors, and therefore the root error detected at line 8 is linked to the previous error value, `OK` in this case.

Explicit guards are used to skip the normal behaviour of the function, as in lines 10 and 17. These guards check if the error variable still contains the `OK` value, and if so, execute the behaviour, otherwise skip it. Note that such guards are also needed in loops containing function calls.

If the error variable contains an error value, this value propagates to the `return` statement, which notifies the callers of the function.

3.4 Handler Determination

Handler determination is *the process of receiving the notification, identifying the exception and determining the associated handler*. The notification of an exception occurs through the use of the `return` statement and catching the returned value in the error variable when invoking a function (lines 4, 11 and 18). This approach is referred to as *explicit stack unwinding*.

The particular exception that occurs is not identified explicitly most of the time, rather a *catch-all* handler is provided. Such handlers are mere guards, that check whether the error value is not equal to `OK`. Typically, such handlers are used to link extra context information to the encountered error (lines 12 − 15), or to clean up allocated resources (lines 20 − 22).

3.5 Resource Cleanup

Resource cleanup is *a mechanism to clean up resources, to keep the integrity, correctness and consistency of the program*.

ASML has no automatic cleanup facilities, although specific handlers typically occur at the end of a function if cleaning up of allocated resources is necessary (lines 20 − 22).

Note that resource cleanup may happen when an exception is raised, and that the cleanup operation itself might give rise to an exception as well. Although not shown in our example for reasons of simplicity, this requires the use of multiple error variables.

3.6 Exception Interface & Reliability Checks

The exception interface *represents the part in a module interface that explicitly specifies the exceptions that might be raised by the module*. ASML uses informal comments to specify which exceptions might be raised by a function.

Consequently, reliability checks that *test for possible faults introduced by the EHM itself* are not possible. The focus of this paper is to analyse which faults can be introduced and to show how they can be detected and prevented.

3.7 Other Components

An EHM consists of several other components than the ones mentioned above. Although these are less important for our purposes, we shortly describe them here for completeness.

Handler scope is *the entity to which an exception handler is attached.* At ASML, handlers have *local* scope: handlers are associated to function calls (lines 12 − 15), where they log extra information, or can be found at the end of a function (lines 20 − 22), where they clean up allocated resources.

Handler binding *attaches handlers to certain exceptions to catch their occurrences in the whole program or part of the program.* ASML uses *semi-dynamic* binding, which means that different handlers can be associated with a single exception in different contexts.

Information passing is defined as *transfer of information useful to the treatment of an exception from its raising context to its handler*. At ASML there is no information passing except for the integer value that is passed to a caller. Although an error log is constructed, the entries are used only for offline analysis.

Criticality management represents *the ability to dynamically change the priority of an exception handler, so that it an be changed based on the importance of the exception, or the importance of the process in which the error occurred.* This is not considered at ASML.

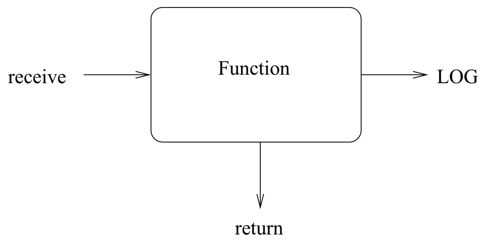


Figure 2: Inputs and outputs of a function with respect to exception handling.

4. A FAULT MODEL FOR EXCEPTION HANDLING

Based on the characterisation presented in the previous section, we establish a fault model for exception handling by means of the return code idiom in this section. The fault model defines when a fault occurs, and includes failure scenarios which explain what happens when a fault occurs.

4.1 General Overview

Our fault model specifies possible faults occurring in a function's implementation of the *exception raising* and *handler determination* components. Those components are clearly the most prone to errors, because their implementation requires a lot of programming work, a good understandability of the idiom, and strict discipline. Although this also holds for the *resource cleanup* component, at ASML this component primarily deals with memory (de)allocation, and we therefore consider it to belong to a *memory handling* concern, for which a different fault model should be established.

The return code idiom at ASML relies on the fact that when an error is received, the corresponding error value should be logged and should propagate to the `return` statement. The programming language constructs that are used to implement this behaviour are function calls, return statements and log calls. The fault model includes a predicate for each of these constructs, and consists of three formulas that specify a faulty combination of these constructs. If one of the formulas is valid for the execution of a function, the EH implementation of the function necessarily contains a fault.

A function is regarded as a black box, i.e., only its input–output behaviour is considered. This perspective allows easy mapping of faults to failure scenarios, at the cost of losing some details due to simplification. Figure 2 gives an overview of the relevant input and outputs of a function. Any error values received from called functions (*receive* predicate) are regarded as input. Outputs are comprised of the error value that is returned by a function (*return*), and values written to the error log (*LOG*). We map the input and outputs to logical predicates as follows.

First, *receive* is a unary predicate that is true for an error value that is received by the function during its execution. For instance, if a function receives an error `PARAM_ERROR` somewhere during its execution, then *receive*(`PARAM_ERROR`) holds true. If a function does not receive an error value during its execution (either because it does not call any functions, or no exception is raised by a called function), then *receive*(OK) holds. Likewise, *return* is a unary predicate that holds true for the error value returned by a function at the end of its execution. Finally, *LOG* is a binary predicate that is true for those two error values (if any) that are written to the error log. The first position of the *LOG* predicate signifies the new error value, while the second position signifies the (old) error to which a link should be established. If and only if nothing is written to the error log during execution, *LOG*(void, void) holds.

The fault model makes two simplifications: it assumes a function receives and logs at most one error during its execution. This is reasonable, because if implemented correctly, no other function should be called once an error value is received. Additionally, if only one error value can be received, it makes little sense to link more than one other error value to it.

4.2 Fault Categories

The fault model consists of three categories, each including a failure scenario, which are explained next. The predicates capturing the faults in each category are displayed in Figure 3. Example code fragments corresponding to Categories 1–3 are displayed in Figures 4–6, respectively.

Category 1. The first category captures those faults where a function raises a new error (y), but fails to perform appropriate logging. There are two cases to distinguish. First, y is considered a root error, i.e., no error has been received from any called function, and therefore $receive(OK)$ holds. The function is thus expected to perform $LOG(y,OK)$. However, a category 1 fault causes the function to perform $LOG(y,z)$ with $z \neq OK$.

Second, y is considered a linked error, i.e., it must be linked to a previously received error x. So, $receive(x)$ holds with $x \neq OK$, and the function is expected to perform $LOG(y,x)$. A category 1 fault in its implementation results in the function performing $LOG(y,z)$ with $x \neq z$.

Category 1 faults have the potential to break error link trees in the error log. The first case causes an error link tree to be improperly initiated, i.e., it does not have the *OK* value at its root. The second case will break (a branch of) an existing link tree, by failing to properly link to the received error value. Furthermore, the faulty *LOG* call will start a new error link tree which has again been improperly rooted. Especially in the latter case it will be hard to recover the chain of errors that occurred, making it neigh impossible to find the root cause of an error.

Category 2. Here the function properly links a new error value y to the received error value x, but then fails to return the new error value (and instead returns z). The calling function will therefore be unaware of the actual exceptional condition, and could therefore have problems determining the appropriate handler. In the special case of $receive(OK)$, the function properly logs a root error y by performing $LOG(y,OK)$, but subsequently returns an error z different from the logged root error y.

Possible problems include corruption of the error log, due to linking to the erroneously returned error value z. Calling functions have no way of knowing the actual value to link to in the error log, because they receive a different error value. Even more seriously, calling functions have no knowledge of the actual error condition and might therefore invoke functionality that may compromise further operation of software or hardware. This problem is most apparent if *OK* is returned while an error has been detected (and logged).

Category 3. The last category consists of function executions that receive an error value x, do not link a new error value to x in the log, but return an error value y that is different from x. The failure scenario is identical to category 2.

Category 1	**Category 2**	**Category 3**
receive(x) $\wedge$	receive(x) $\wedge$	receive(x) $\wedge$
LOG(y,z) $\wedge$	LOG(y,x) $\wedge$	LOG$(void, void)$ $\wedge$
$x \neq z$	return(z) $\wedge$	return(y) $\wedge$
	$y \neq z$	$x \neq y$

Figure 3: Predicates for the three fault categories.

5. SMELL: STATICALLY DETECTING EXCEPTION HANDLING FAULTS

Based on the fault model we developed SMELL, the *State Machine for Error Linking and Logging*, which is capable of statically detecting violations to the return code idiom in the source code, and is implemented as a CodeSurfer[1] plugin. We want to detect faults statically, instead of through testing as is usual for fault models, because early detection and prevention of faults is less costly [3, 6], and because testing exception handling is inherently difficult.

5.1 Implementation

SMELL statically analyses executions of a function in order to prove the truth of any one of the logic formulas of our fault model. The analysis is static in the sense that no assumptions are made about the inputs of a function. Inputs consist of formal or global variables, or values returned by called functions.

We represent an execution of a function by a finite path through its control-flow graph. Possibly infinite paths due to recursion or iteration statements are dealt with as follows. First, SMELL performs an intra-procedural analysis only, i.e., calls to other functions are not followed, and therefore recursion is no problem during analysis. Second, loops created by iteration statements are dealt with by caching analysis results at each node of the control-flow graph. We discuss this mechanism later.

The analysis performed by SMELL is based on the evaluation of a deterministic (finite) state machine (SM) during the traversal of a path through the control-flow graph. The SM inspects the properties of each node it reaches, and then changes state accordingly. A fault is detected if the SM reaches the *reject* state; conversely, a path is free of faults if the SM reaches the *accept* state.

The error variable is a central notion in the current implementation of SMELL. An error variable, such as the `r` variable in Figure 1, is used by a programmer to keep track of previously raised errors. SMELL attempts to identify such variables automatically based on a number of properties. Unfortunately, the idiom used for exception handling does not specify a naming convention for error variables. Hence, each programmer picks his or her favourite variable name, ruling out a simple lexical identification of these variables. Instead, a variable qualifies as an error variable in case it satisfies the following properties:

- it is a local variable of type `int`,
- it is assigned only constant (integer) values or function call results,
- it is not passed to a function as an actual, unless in a log call,
- no arithmetic is performed using the variable.

Most functions in the ASML source base use at most one error variable, but in case multiple are used, SMELL considers each control-flow path separately for each error variable. Functions for which no error variable can be identified are not considered for further analysis. We discuss the limitations of this approach at the end of this section.

Describing the complete SM would require too much space. Therefore we limit our description to the states defined in the SM, and show a subset of the transitions by means of example runs.

The following states are defined in the SM:

Accept and **Reject** represent the absence and presence of a fault on the current control-flow path, respectively.

Entry is the start state, i.e., the state of the SM before the evaluation of the first node. A transition from this state only occurs when an initialisation of the considered error variable is encountered.

OK reflects that the current value of the error variable is the OK constant. Conceptually this state represents the absence of an exceptional condition.

Not-OK is the converse, i.e., the error variable is known to be anything but OK, though the exact value is not known. This state can be reached when a path has taken the true branch of a guard like `if(r != OK)`.

Unknown is the state reached if the result of a function call is assigned to the error variable. Due to our limitation to intra-procedural analysis, we conservatively assume function call results to be unknown.

Constant is a parametrised state that contains the constant value assigned to the error variable. This state can be reached after the assignment of a literal constant value to the error variable.

All states also track the error value that was last written to the log file. This information is needed to detect faults in the logging of errors.

While traversing paths of the control-flow graph of a function, the analysis caches results in order to prevent infinite traversals of loops and to improve efficiency by eliminating redundant computations. In particular, the state (including associated values of parameters) in which the SM reaches each node is stored. The analysis then makes sure that each node is visited at most once given a particular state. The same technique is used by Engler *et al.* in [13].

5.2 Example Faults

The following three examples show how the SM detects faults from each of the categories in the fault model. States reached by the SM are included in the examples as comments, and where appropriate the last logged error value is mentioned in parentheses. First, consider the code snippet in Figure 4. A fault of category 1 possibly occurs on the path that takes the true branch of the `if` statement on line 4. If the function call at line 3 returns with an error value, say `INIT_ERROR` then receive(`INIT_ERROR`) holds. The call to the LOG function on line 5 makes LOG(`RANGE_ERROR`, `OK`) true,

[1] www.grammatech.com

```
1  int calibrate(int a) {      // Entry
2    int r = OK;               // OK
3    r = initialise();         // Unknown
4    if(a == 1)
5      LOG(RANGE_ERROR, OK);   // Reject
6    ...
7  }
```

Figure 4: Example of fault category 1.

and since OK is different from INIT_ERROR, all clauses of the predicate for category 1 are true, resulting in a fault of category 1.

SMELL detects this fault as follows, starting in the Entry state on line 1. The initialisation of r, which has been identified as an error variable, causes a transition to the OK state on line 2. The assignment to r of the function call result on line 3 results in the Unknown state. On the true branch of the if statement on line 4, a (new) root error is raised. The cause of the fault lies here. SMELL reaches the Reject state at line 5 because if an error value (say INIT_ERROR) would have been returned from the call to the initialise function, it is required to link the RANGE_ERROR to the INIT_ERROR, instead of linking to OK.

```
1  int align() {               // Entry
2    int r = OK;               // OK
3    r = initialise();         // Unknown
4    if(r != OK)               // Not-OK
5      LOG(ALIGN_ERROR, r); // Not-OK (ALIGN_ERROR)
6    return r;                 // Reject
7  }
```

Figure 5: Example of fault category 2.

The function in Figure 5 exhibits a fault of category 2 on the path that takes the true branch of the if statement. Again, suppose receive(INIT_ERROR) holds, then the function correctly performs LOG(ALIGN_ERROR, INIT_ERROR). The fault consists of the function not returning ALIGN_ERROR, but INIT_ERROR, because after linking to the received error, the new error value is not assigned to the error variable.

Again SMELL starts in the Entry state, and subsequently reaches the OK state after the initialisation of the error variable r. The initialise function is called at line 3, and causes SMELL to enter the Unknown state. Taking the true branch at line 4 implies that the value of r must be different from OK, and SMELL records this by changing to the Not-OK state. At line 5 an ALIGN_ERROR is linked to the error value currently stored in the r variable. SMELL then reaches the return statement, which causes the error value to be returned that was returned from the initialise function call at line 3. Since the returned value differs from the logged value at this point, SMELL transits to the Reject state.

Category 3 faults are similar to category 2, but without any logging taking place. Suppose again that for the function in Figure 6 receive(INIT_ERROR) holds. For the path taking the true branch of the if statement a value different from INIT_ERROR will be returned, i.e., PROCESS_ERROR.

Until the assignment at line 5 the SM traverses through the same sequence of states as for the previous examples. However, the assignment at line 5 puts SMELL in the Reject state, because the previously received error value has been overwritten. A category 3 fault is therefore manifest.

5.3 Fault Reporting

```
1  int process(int a) {    // Entry
2    int r = OK;           // OK
3    r = initialise();     // Unknown
4    if(a == 2) {
5      r = PROCESS_ERROR; // Reject
6    }
7    ...
8    return r;
9  }
```

Figure 6: Example of fault category 3.

SMELL reports the presence of faults using a more fine grained view of the source code than the fault model. While the fault model takes a black box perspective, i.e., regarding behaviour only at the interface level, SMELL reports detected faults using a white box perspective, i.e., considering the implementation level details of a function. The white box perspective is considered to be more useful when interpreting actual fault reports, which developers may have to process.

In the following we present a list of “low-level faults”, or programmer mistakes, that SMELL reports to its users. For each programmer mistake we mention here the associated fault categories from the fault model. SMELL itself does not report these categories to the user. To help users interpreting the reported faults, SMELL prints the control-flow path leading up to the fault, and the associated state transitions of the SM.

function does not return occurs when a function declares and uses an error variable (i.e., assigns a value to it), but does not return its value. If present, SMELL detects this fault at the return statement of the function under consideration. This can cause category 2 or 3 faults.

wrong error variable returned occurs when a function declares and uses an error variable but returns another variable, or when it defines multiple error variables, but only returns one of them and does not link the others to the returned one in the appropriate way. This can cause category 2 or 3 faults.

assigned and logged value mismatch occurs when the error value that is returned by a function is not equal to the value last logged by that function. This can cause category 2 faults.

not linked to previous value occurs when a LOG call is used to link an error value to a previous value, but this latter value was not the one that was previously logged. If present, SMELL detects this fault at the call site of the log function. This causes category 1 faults.

unsafe assignment occurs when an assignment to an error variable overwrites a previously received error value, while the previous error value has not yet been logged. Clearly, if present SMELL detects this fault at the assignment that overwrites the previous error value.

5.4 Limitations

Our approach is both formally unsound and incomplete, which is to say that our analysis proves neither the absence nor the presence of ‘true’ faults. In other words, both false negatives (missed faults) or false positives (false alarms) are possible. False negatives for example occur when SMELL detects a fault on a particular control-flow path, and stops traversing that path. Consequently, faults occurring later in the path will go unnoticed. The unsoundness property and incompleteness properties do not necessarily harm the usefulness of our tool, given that the tool still allows us to detect a large number of faults that may cause much machine down-time, and that the number of false positives remains manageable. The experimental results (see Section 6) show that we are currently within acceptable margins.

SMELL also exhibits a number of other limitations:

Meta assignments Meta assignments are assignments involving two different error variables, such as `r = r2;`. SMELL does not know how to deal with such statements, since it traverses the control-flow paths for each error variable separately. As a result, when considering the `r` variable, SMELL does not know what the current value of `r2` is, and vice versa.

For the moment, SMELL recognises such statements and simply stops traversing the current control-flow path.

Variableless log calls Variableless log calls are calls to the `LOG` function that do not use an error variable as one of their actual arguments, but instead only use constants, such as for example `LOG(PARAM_ERROR,OK)`.

The problem with such calls appears when a function defines more than one error variable. Although a developer is able to tell which error variable is considered from the context of the call, SMELL has trouble associating the call to a specific error variable.

Whenever possible, SMELL tries to recover from such calls intelligently. For example, in the following case:

```
r =PARAM_ERROR;
LOG(PARAM_ERROR,OK);
```

SMELL is able to infer that the log call belongs to the `r` variable, because it logs the constant that is assigned to that variable. However, the problem reappears when a second error variable is considered. When checking that variable and encountering the `LOG` call, SMELL will report an error if the error value contained in the second error variable differs from the logged value, because it does not know the `LOG` call belongs to a different error variable.

Infeasible Paths Infeasible paths are paths through the control-flow graph that can never occur at runtime, but that are considered as valid paths by SMELL. SMELL only considers the values for error variables, and smartly handles guards involving those variables. But it does not consider any other variables, and as such cannot infer, for example, that certain conditions using other variables are in fact mutually exclusive.

Wrong Error Variable Identification The heuristic SMELL uses to identify error variables is not perfect. False positives occur when integer values are used to catch return values from library functions, for example, such as `puts` or `printf`. Additionally, false negatives occur when developers pass the error variable as an actual or perform some arithmetic operations on it. This is not allowed by the ASML coding standard, however.

Currently, false positives are easily identified manually, since SMELL's output reports which error variable was considered. If this error variable is meaningless, inspection of the fault can safely be skipped.

6. EXPERIMENTAL RESULTS

6.1 General Remarks

Table 1 presents the results of applying SMELL on 5 relatively small ASML components. The first column lists the component that was considered together with its size, column 2 lists the number of faults reported by SMELL, column 3 contains the number of false positives we manually identified among the reported faults, column 4 shows the number of SMELL limitations that are encountered, and finally column 5 contains the number of validated faults, or 'true' faults.

	reported	false positives	limitations	validated
CC1 (3 kLoC)	32	2	4	26
CC2 (19 kLoC)	72	20	24	28
CC3 (15 kLoC)	16	0	3	13
CC4 (14.5 kLoC)	107	14	13	80
CC5 (15 kLoC)	9	0	3	6
total (66.5 kLoC)	236	36	47	153

Table 1: Reported number of faults by SMELL for five components.

Four of the five components are approximately of the same size, but there is a striking difference between the numbers of *reported* faults. The number of reported faults for the CC3 and CC5 components are much smaller than those reported for the CC2 and CC4 components. When comparing the number of *validated* faults, the CC4 component clearly stands out, whereas the number for the other three components is approximately within the same range.

Although the CC1 component is the smallest one, its number of validated faults is large compared to the larger components. This is due to the fact that a heavily-used macro in the CC1 component contains a fault. Since SMELL is run after macro expansion, a fault in a single macro is reported at every location where that macro is used.

The number of validated faults reported for the CC5 component is also interestingly low. This component is developed by the same people responsible for the EHM implementation. As it turns out, even these people violate the idiom from time to time, which shows that the idiom approach is difficult to adhere to. However, it is clear that the CC5 code is of better quality than the other code.

Overall, we get 236 reported faults, of which 47 (20%) are reported by SMELL as a limitation. The remaining 189 faults were inspected manually, and we identified 36 false positives (15% of reported faults). The remaining 153 faults are thus validated, or in other words, we found 2.3 true faults per thousand lines of code.

6.2 Fault Distribution

A closer look at the 153 validated faults shows that 13 faults are due to a function not returning, 28 due to the wrong error variable being returned, 68 due to unsafe assignments, 11 due to incorrect logging, and 42 due to an assigned and logged value mismatch.

The *unsafe assignment* fault occurs when the error variable contains an error value that is subsequently overwritten. This kind of fault is by far the one that occurs the most (68 out of 153 = 44%), followed by the *assigned and logged value mismatch* (42 out of 153 = 27%). If we want to minimise the exception handling faults, we should develop an alternative solution that deals with these two kinds of faults.

Accidental overwriting of the error value typically occurs because the control flow transfer when the exception is raised is not implemented correctly. This is mostly due to a forgotten guard that involves the error variable ensuring that normal operation only continues when no exception has been reported previously. An example of such a fault is found in Figure 6.

The second kind of fault occurs in two different situations. First, as exemplified in Figure 5, when a function is called and an exception is received, a developer might link an exception to the received one, but forgets to assign the linked exception to the error variable. Second, when a root error is detected and a developer assigns the appropriate error value to the error variable, he might forget to log that value.

6.3 False positives

The number of false positives is sufficiently low to make SMELL useful in practice. A detailed look at these false positives reveals the reasons why they occur and allows us to identify where we can improve SMELL.

Of the 36 false positives identified, 23 are due to an incorrect identification of the error variable, 7 are due to SMELL getting confused when multiple error variables are used, 4 occur because an infeasible path has been followed, and 2 false positives occur due to some other (mostly domain-specific) reason.

These numbers indicate that the largest gain can be obtained by improving the error variable identification algorithm, for example by trying to distinguish ASML error variables from “ordinary” error variables. Additionally, they show that the issue of infeasible paths is not really a large problem in practice.

7. AN ALTERNATIVE EXCEPTION HANDLING APPROACH

In order to reduce the number of faults in exception handling code, alternative approaches to exception handling should be studied. A solution which introduces a number of simple macros has been proposed by ASML, and we will discuss it here. We thereby keep in mind that we know that the two most frequently occurring faults are overwriting of the error value and the mismatch between the value assigned to the error variable and the value actually logged.

The solution is based on two observations.

First, it encourages developers to no longer write assignments to the error variable explicitly, and it manages them automatically inside the macros. Such assignments can either be constant assignments, when declaring a root error, or function-call assignments, when calling a function. By embedding such assignments inside specific macros and surrounding them with appropriate guards, we can prevent accidental overriding of error values.

Second, the macros ensure that assignments are accompanied by the appropriate LOG calls, in order to avoid a mismatch between logged and assigned values. As explained in the previous section, such a mismatch occurs when declaring a root error or when linking to a received error. Consequently, we introduce a ROOT_LOG and LINK_LOG macro that should be used in those situations and that take care of all the work.

The proposed macro’s are defined in Figure 7. The ROOT_LOG macro should be used whenever a root error is detected, while the LINK_LOG macro is used when calling a function and additional information can be provided when an error is detected. Additionally, a NO_LOG macro is introduced that should be used when calling a function and not linking extra information if something goes wrong.

Using these macros, the example code from Section 3 is changed into the code that can be seen in Figure 8.

It is interesting to observe that using these macros drastically reduces the number of (programmer visible) control-flow branches. This not only improves the function’s understandability and maintainability, but also causes a significant drop in code size, if we consider that the return code idiom is omnipresent in the ASML code base. Moreover, the exception handling code is separated from the ordinary code, which allows the two to evolve separately. More research is needed to study these advantages in detail.

The solution still exhibits a number of drawbacks.

First of all, the code that cleans up memory resources remains as is. This is partly due to the fact that we did not focus on such code, since we postulate that it belongs to a different concern. However, such code also differs significantly between different functions and source components, which makes it harder to capture it into a set of appropriate macros.

```
#define ROOT_LOG(error_value, error_var)\
      error_var = error_value;\
      LOG(error_value, OK);
```

```
#define LINK_LOG(function_call, error_value, error_var)\
   if(error_var == OK) {\
      int _internal_error_var = function_call;\
      if(_internal_error_var != OK) {\
         LOG(error_value, _internal_error_var);\
         error_var = error_value;\
      }\
   }
```

```
#define NO_LOG(function_call, error_var)\
   if(error_var == OK) \
      error_var = function_call;
```

Figure 7: Definitions of proposed exception handling macro’s.

Second, reliability checks are still not available. It remains the developer’s responsibility to use the macros in the correct way, by passing the correct arguments in the correct way. When this is not the case, for example because arguments are reversed, faults of category 1 will occur. Given that we detected only a small fraction of faults of this category, we believe this will not pose serious problems.

Last, the macros do not tackle faults that concern the returning of the appropriate error value. Since this was a deliberate choice, because such errors are rather scarce and can be easily found, this comes as no surprise.

```
int f(int a, int* b) {
   int r = OK;
   bool allocated = FALSE;
   r = mem_alloc(10, (int *)b);
   allocated = (r == OK);
   if((a < 0) || (a > 10))
       ROOT_LOG(PARAM_ERROR,r);
   LINK_LOG(g(a),LINKED_ERROR,r);
   NO_LOG(h(b), r);
   if((r != OK) && allocated)
      mem_free(b);
   return r;
}
```

Figure 8: Function f implemented by means of the alternative macros.

8. DISCUSSION

In our examples, we found 2.3 deviations from the return code idiom per 1000 lines of code. In this section, we discuss some of the implications of this figure, looking at questions such as the following: How does the figure relate to reported defect densities in other systems? What, if anything, does the figure imply for system reliability? What does the figure teach us on idiom and coding standard design?

8.1 Representativeness

A first question to be asked is to what extent our findings are representative for other systems. The software under study has the following characteristics:

- It is part of an embedded system in which proper exception handling is essential: The system consists of hundreds of sensors, actuators and other hardware components, all of which can fail in various ways. The software must be capable of handling such exceptions appropriately.
- Exception handling is implemented using the return code idiom for which little or no automated tool support is used (beyond standard `lint`-like facilities).
- Before release, the software components in question are subjected to a thorough code review.
- The software is subjected to rigorous unit, integration, and system tests.

In other words, we believe our findings hold for software that is the result of a state-of-the-art development process.

The reason that we find so many exception handling faults in spite of this state-of-the-art process is that current ways of working are not effective in finding such faults: tool support is inadequate, regular reviews tend to be focused on "good weather behaviour" — and even if they are aimed at exception handling faults these are too hard to find, and testing exception handling is notoriously hard.

8.2 Defect Density

What meaning should we assign to the value of 2.3 exception handling faults per 1000 lines of code (kLoC) we detected?

It is tempting to compare the figure to reported defect densities. For example, an often cited paper reports a defect density between 5 and 10 per kLoC for software developed in the USA and Europe [11]. More recently, in his ICSE 2005 state-of-the-art report, Littlewood states that studies show around 30 faults per kLoC for commercial systems [23].

There are, however, several reasons why making such comparisons is questionable, as argued, for example, by [15]. First, there is neither consensus on what constitutes a defect, nor on the best way to measure software size in a consistent and comparable way. In addition to that, defect density is a product measure that is derived from the process of finding defects. Thus, "defect density may tell us more about the quality of the defect finding and reporting process than about the quality of the product itself" [15, p.346]. This particularly applies to our setting, in which we have adopted a new way to search for faults.

The consequence of this is that no conclusive statement on the relative defect density of the system under study can be made. We cannot even say that our system is of poorer quality than another with a lower reported density, as long as we do not know whether the search for defects included a hunt for idiom errors similar to our approach.

What we can say, however, is that a serious attempt to determine defect densities should include an analysis of the faults that may arise from idioms used for dealing with crosscutting concerns. Such an analysis may also help when attempting to explain observed defect densities for particular systems.

8.3 Reliability

We presently do not know what the likelihood is that an exception handling fault actually leads to a failure, such as an unnecessary halt, an erroneously logged error value, or the activation of the wrong exception handler. As already observed by Adams in 1984, more faults need not lead to more failures [1]. We are presently investigating historical system data to clarify the relation between exception handling faults and their corresponding failures. This, however, is a time consuming analysis requiring substantial domain knowledge in order to understand a problem report, the fault identified for it (which may have to be derived from the fix applied) and to see their relation to the exception handling idiom.

8.4 Idiom design

The research we are presenting is part of a larger, ongoing effort in which we are investigating the impact of crosscutting concerns on embedded C code [5, 4]. The traditional way of dealing with such concerns is by devising an appropriate coding idiom. What implications do our findings have on the way we actually design such coding idioms?

One finding is that an idiom making it too easy to make small mistakes can lead to many faults spread all over the system. For that reason, idiom design should include the step of constructing an explicit fault model, describing what can go wrong when using the idiom. This will not only help in avoiding such errors, but may also lead to a revised design in which the likelihood of certain types of errors is reduced.

A second lesson to be drawn is that the possibility to check idiom usage automatically should be taken into account: static checking should be designed into the idiom. As we have seen, this may require complex analysis at the level of the program dependence graph as opposed to the (elementary) abstract syntax tree.

9. CONCLUDING REMARKS

Contributions

Our contributions are summarised as follows. First, we provided empirical data about the use of an exception handling mechanism based on the return code idiom in an industrial setting. This data shows that the idiom is particularly error prone, due to the fact that it is omnipresent as well as highly tangled, and requires focused and well-thought programming. Second, we defined a series of steps to regain control over this situation, and answer the specific questions we raised in the introduction. These steps consist of the characterisation of the return code idiom in terms of an existing model for exception handling mechanisms, the construction of a fault model which explains when a fault occurs in the most error prone components of the characterisation, the implementation of a static checker tool which detects faults as predicted by the fault model, and the introduction of an alternative solution, based on experimental findings, which is believed to remove the faults most occurring.

We feel these contributions are not only a first step toward a reliability check component for the return code idiom, but also provide a good basis for (re)considering exception handling approaches when working with programming languages without proper exception handling support. We showed that when designing such idiom-based solutions, a corresponding fault model is a necessity to assess the fault-proneness, and the possibility of static checking should be seriously considered.

Future work

There are several ways in which our work can be continued:

- apply SMELL to more ASML components, in order to perform more extensive validation. Additionally, some components already use the macros presented in Section 7, which allows us to compare the general approach to the alternative approach, and assess benefits and possible pitfalls in more detail. We initiated such efforts, and are currently analysing approximately two million lines of C code for this.

- apply SMELL to non-ASML systems, such as open-source systems, in order to generalise it and to present the results openly.
- apply SMELL to other exception handling mechanisms for C, such as those based on the `setjmp/longjmp` idiom, to analyse which approach is most suited.
- investigate aspect-oriented opportunities for exception handling, since benefits in terms of code quality can be expected if exception handling behaviour is completely separated from ordinary behaviour [22]. Furthermore, such an approach may help to make the exception interface (see Section 3.6) explicit, similar to the domain-specific language we use to specify parameter declarations in [4].

Acknowledgements

This work has been carried out as part of the Ideals project under the auspices of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program. The authors would like to thank Remco van Engelen, Christian Bakker, Michiel Kamps and Pieter ten Pierick for their help with and comments on SMELL.

10. REFERENCES

[1] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[2] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM, January 2002.

[3] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[4] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating Idiomatic Crosscutting Concerns. In *Proceedings of the 21th International Conference on Software Maintenance (ICSM), to appear*. IEEE Computer Society, 2005.

[5] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'04)*, pages 200–209. IEEE Computer Society Press, September 2004.

[6] M. Bush. Improving software quality: the use of formal inspections at the jpl. In *Proceedings of the 12th international conference on Software engineering (ICSE)*, pages 196–199. IEEE Computer Society, 1990.

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[8] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244. ACM, November 2002.

[9] F. Christian. *Exception handling and tolerance of software faults*, chapter 4, pages 81–107. John Wiley & Sons, 1995.

[10] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 57–68. ACM, May 2002.

[11] M. Dyer. The cleanroom approach to quality software development. In *Proceedings of the 18th International Computer Measurement Group Conference*, pages 1201–1212. Computer Measurement Group, 1992.

[12] E. v. Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE Computer Society, 2002.

[13] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 1–16. USENIX Association, October 2000.

[14] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 44–53. ACM, May 1996.

[15] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A rigorous and Practical Approach*. PWS Publishing Company, second edition, 1997.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245. ACM, May 2002.

[17] N. H. Gehani. Exceptional C or C with exceptions. *Software Practice and Experience*, 22(10):827–848, 1992.

[18] S. Johnson. Lint, a C Program Checker. Technical Report 65, Bell Laboratories, Dec. 1977.

[19] J. Lang and D. B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(2):274 – 301, Mar. 1998.

[20] P. A. Lee. Exception handling in C programs. *Software Practice and Experience*, 13(5):389–405, 1983.

[21] J.-L. Lions. Ariane 5 flight 501 failure. Technical report, ESA/CNES, 1996.

[22] M. Lippert and C. V. Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22th international conference on Software engineering (ICSE)*, pages 418 – 427. IEEE Computer Society, 2000.

[23] B. Littlewood. Dependability assessment of software-based systems: state of the art. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 6–7, New York, NY, USA, 2005. ACM Press.

[24] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *5th Symposium on Operating System Design and Implementation (OSDI'02)*. USENIX Association, December 2002.

[25] S. Paul and A. Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[26] E. S. Roberts. Implementing exceptions in C. Technical Report 40, Digital Systems Research Center, 1989.

[27] M. Robillard and G. C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Department of Computer Science, University of British Columbia, 1999.

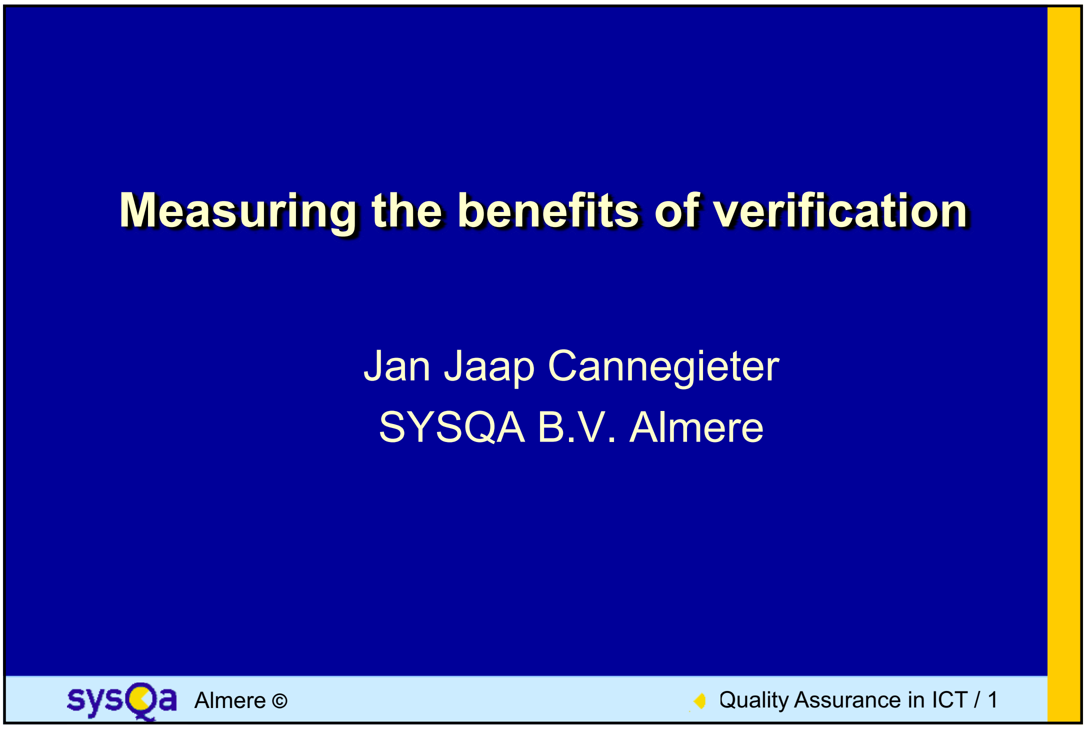
[28] M. P. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, 2003.

[29] T. Tourwé and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 91 – 100. IEEE Computer Society, 2003.

[30] W. N. Toy. Fault-tolerant design of local ess processors. In *Proceedings of IEEE*, pages 1126–1145. IEEE Computer Society, 1982.

[31] H. Winroth. Exception handling in ANSI C. Technical Report ISRN KTH NA/P–93/15–SE, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, 1993.

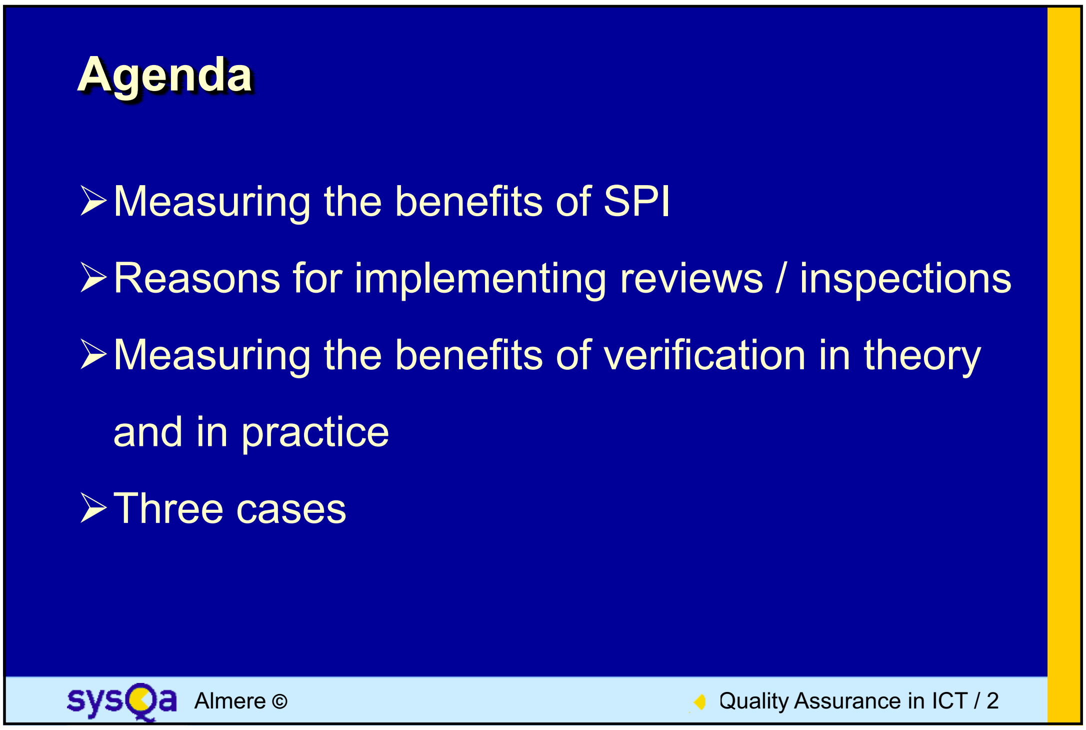
[32] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *6th Symposium on Operating System Design and Implementation (OSDI'04)*, pages 273–288. USENIX Association, December 2004.



Measuring the benefits of verification

Jan Jaap Cannegieter
SYSQA B.V. Almere

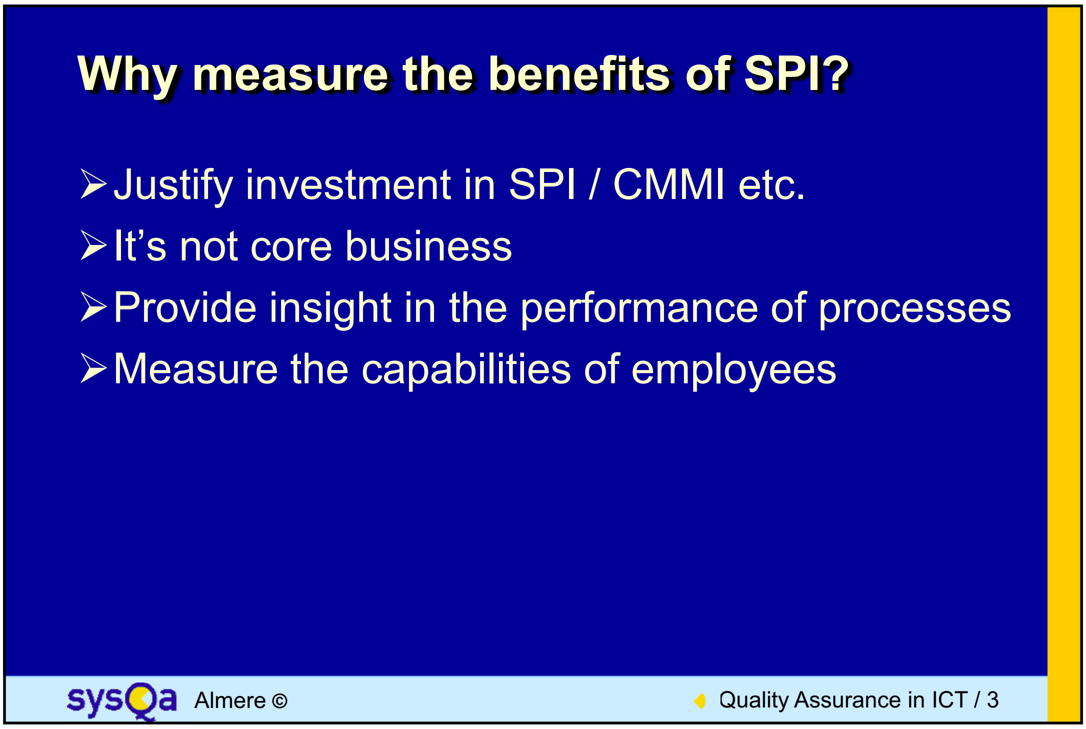
sysqa Almere © Quality Assurance in ICT / 1



Agenda

- ➢ Measuring the benefits of SPI
- ➢ Reasons for implementing reviews / inspections
- ➢ Measuring the benefits of verification in theory and in practice
- ➢ Three cases

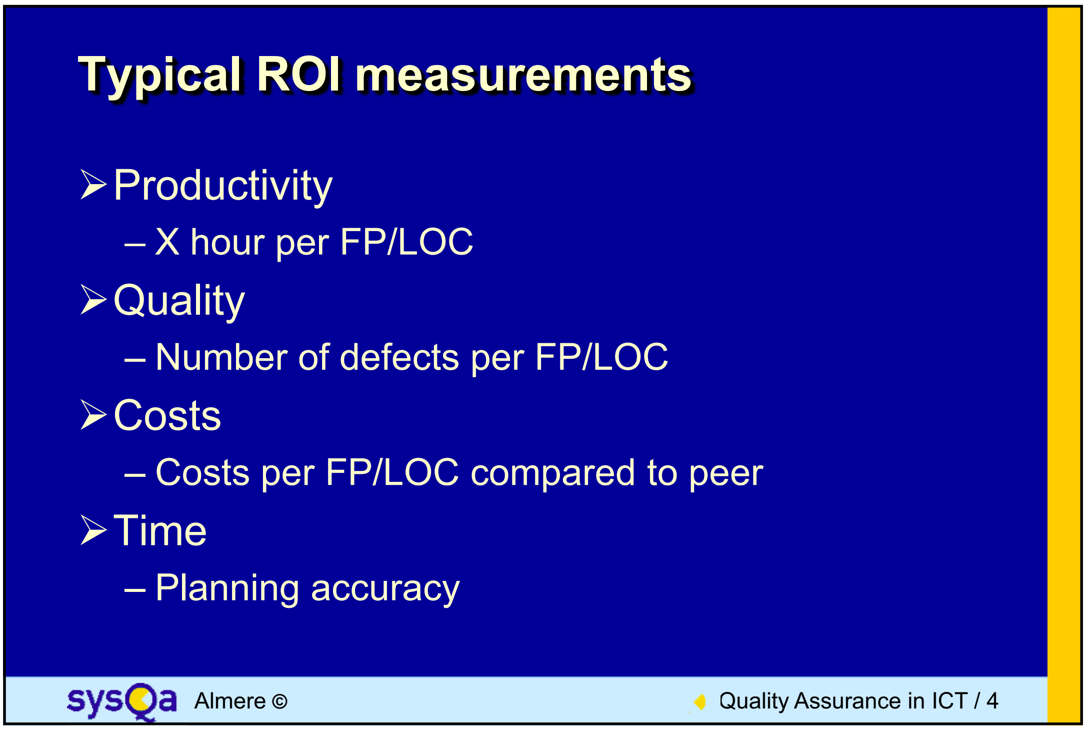
sysqa Almere © Quality Assurance in ICT / 2



Why measure the benefits of SPI?

- Justify investment in SPI / CMMI etc.
- It’s not core business
- Provide insight in the performance of processes
- Measure the capabilities of employees

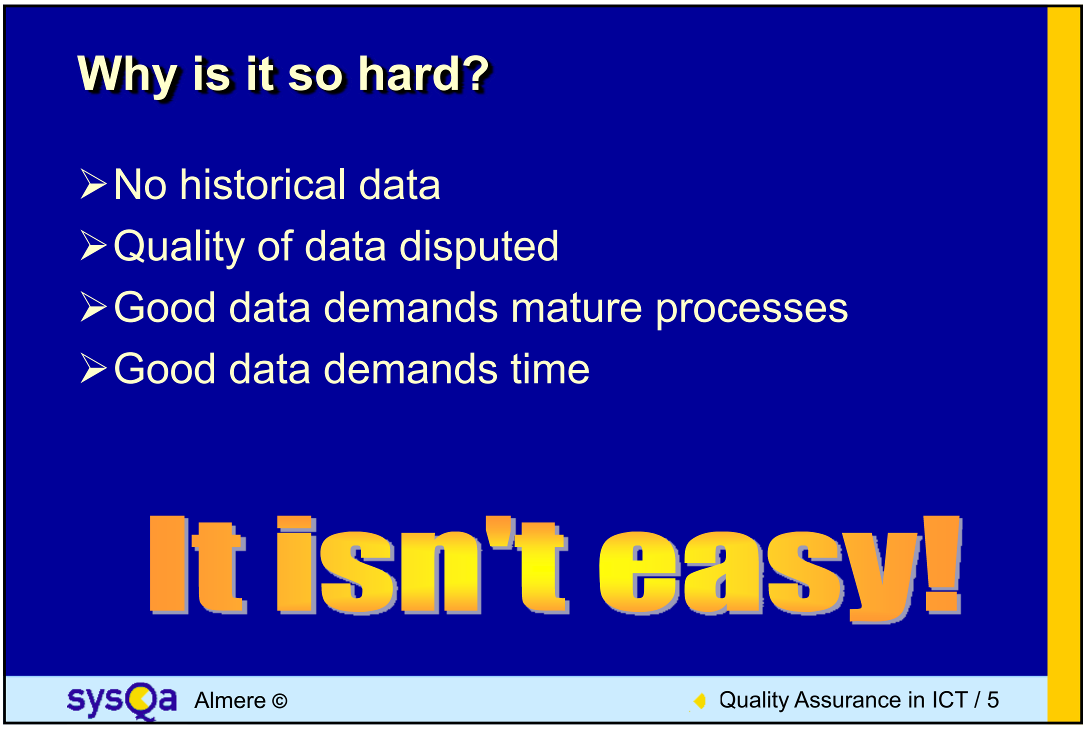
sysQa Almere © Quality Assurance in ICT / 3



Typical ROI measurements

- Productivity
  - X hour per FP/LOC
- Quality
  - Number of defects per FP/LOC
- Costs
  - Costs per FP/LOC compared to peer
- Time
  - Planning accuracy

sysQa Almere © Quality Assurance in ICT / 4



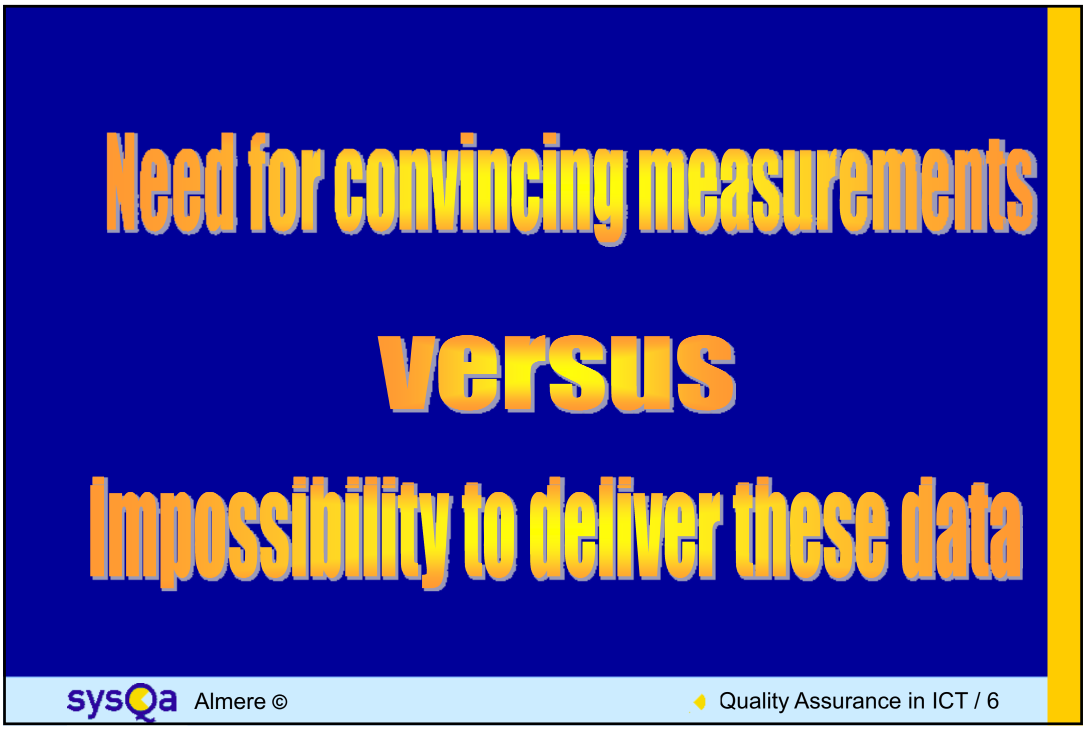
Why is it so hard?

- No historical data
- Quality of data disputed
- Good data demands mature processes
- Good data demands time

It isn't easy!

sysQa Almere ©

Quality Assurance in ICT / 5



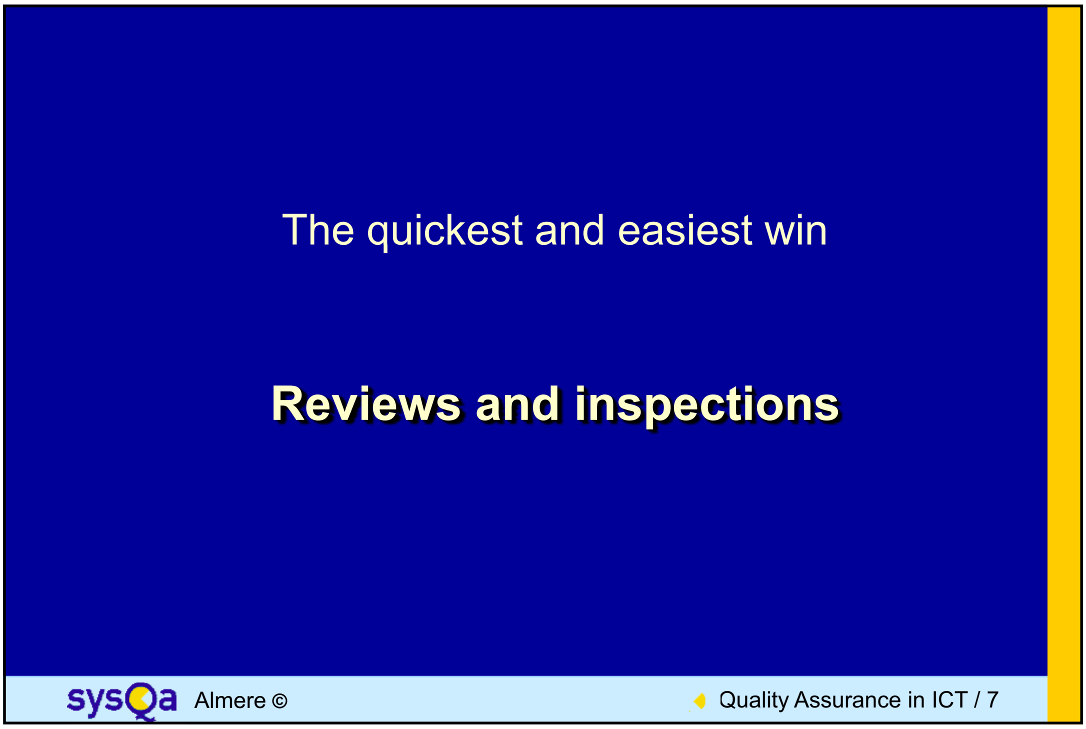
Need for convincing measurements

versus

Impossibility to deliver these data

sysQa Almere ©

Quality Assurance in ICT / 6

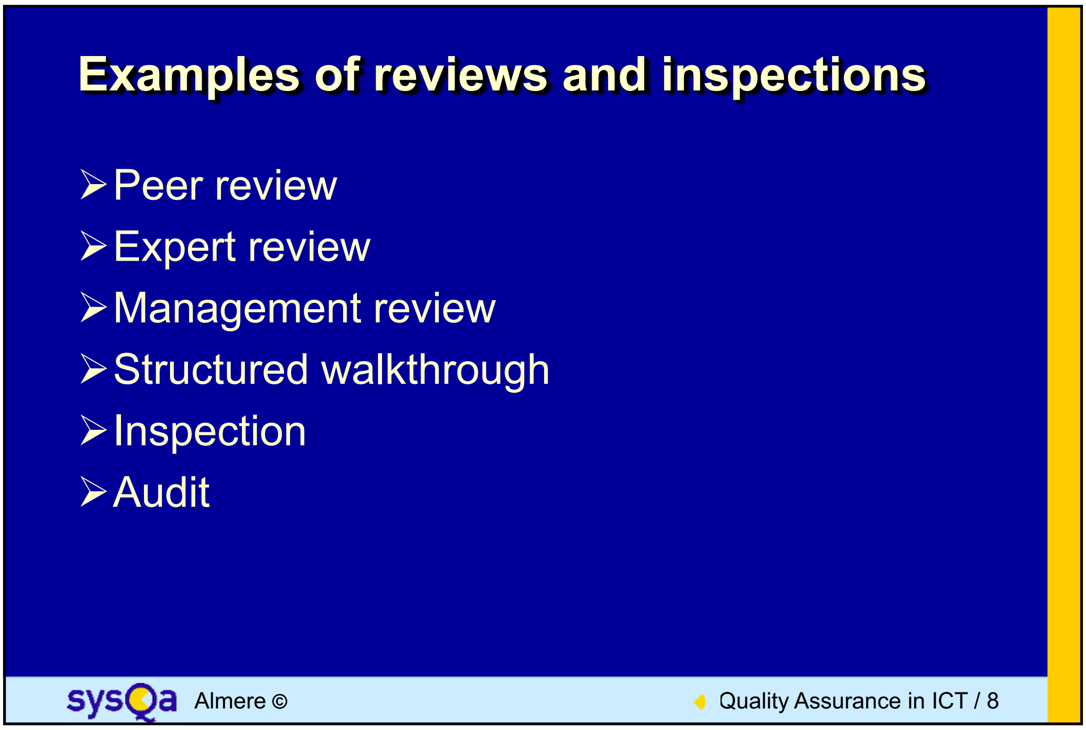


The quickest and easiest win

Reviews and inspections

Almere ©

Quality Assurance in ICT / 7

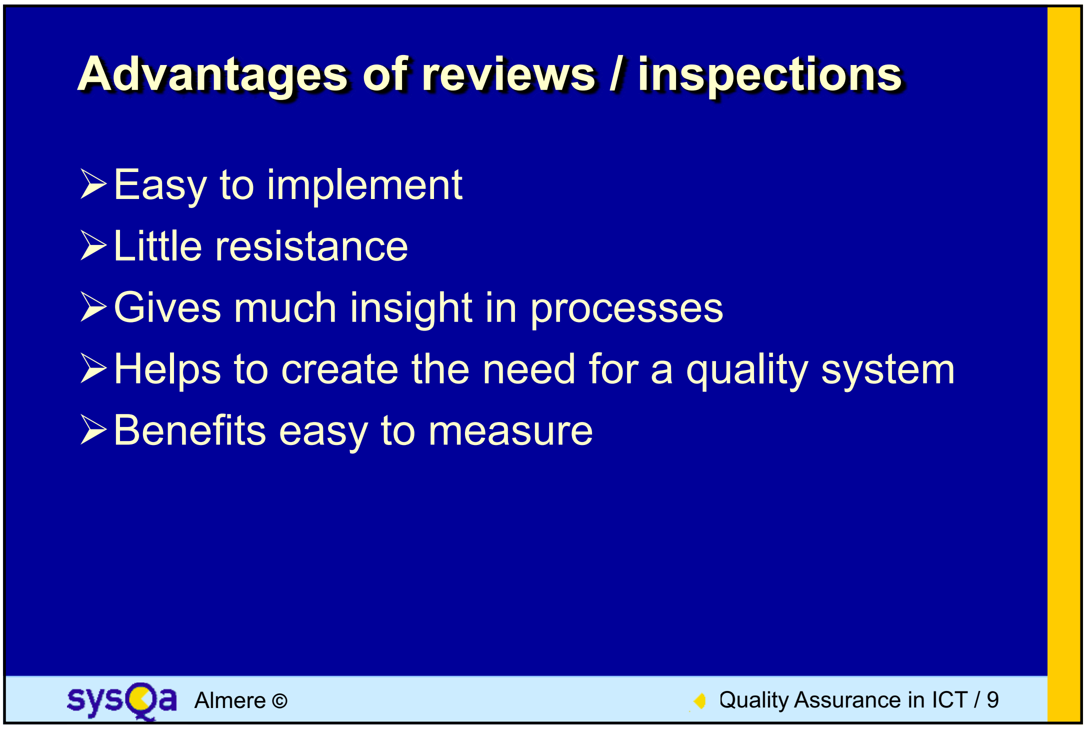


Examples of reviews and inspections

- Peer review
- Expert review
- Management review
- Structured walkthrough
- Inspection
- Audit

Almere ©

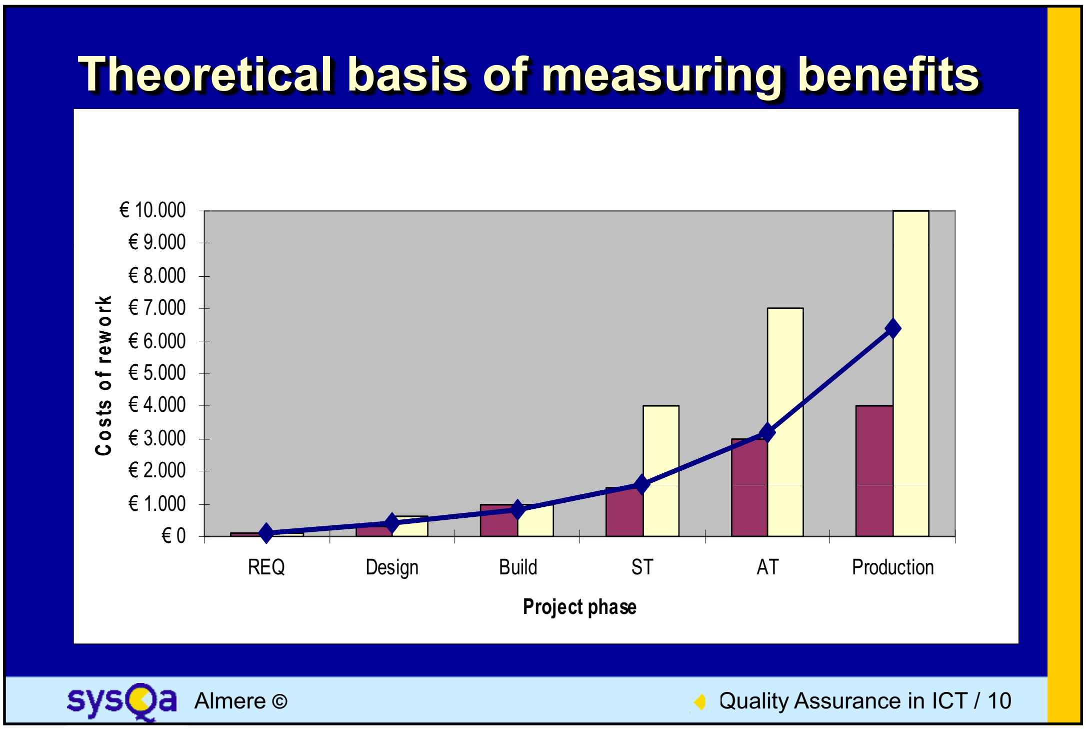
Quality Assurance in ICT / 8



Advantages of reviews / inspections

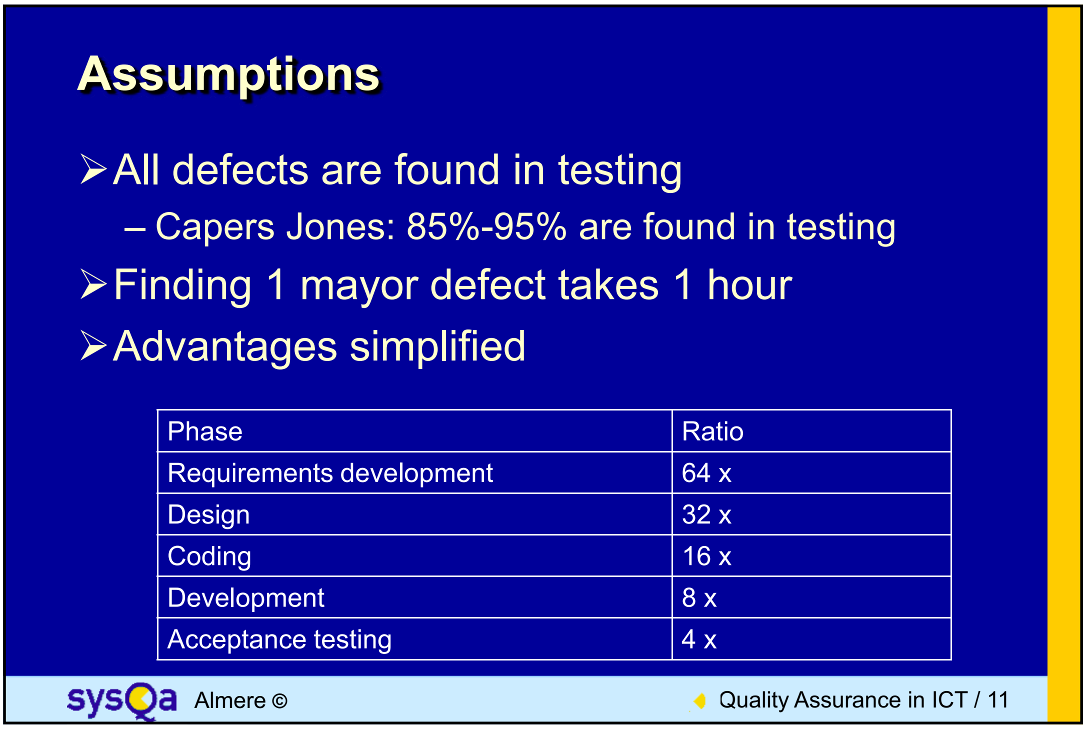
- Easy to implement
- Little resistance
- Gives much insight in processes
- Helps to create the need for a quality system
- Benefits easy to measure

sysQa Almere © Quality Assurance in ICT / 9



Theoretical basis of measuring benefits

sysQa Almere © Quality Assurance in ICT / 10



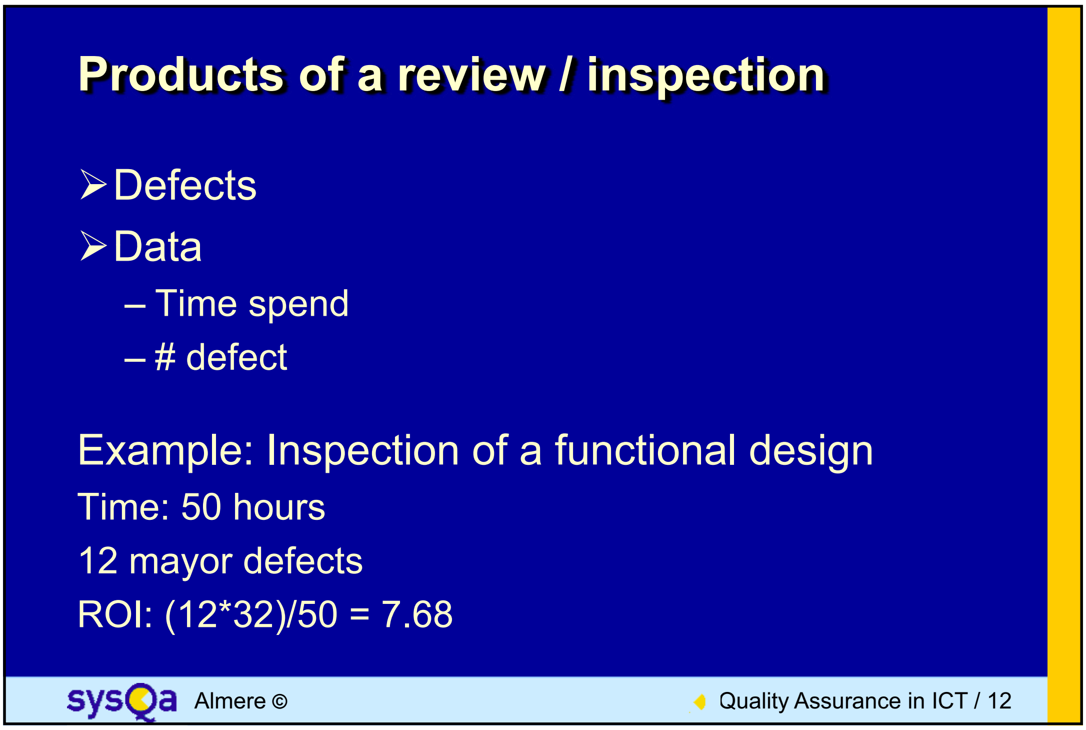
Assumptions

- All defects are found in testing
  - Capers Jones: 85%-95% are found in testing
- Finding 1 mayor defect takes 1 hour
- Advantages simplified

Phase	Ratio
Requirements development	64 x
Design	32 x
Coding	16 x
Development	8 x
Acceptance testing	4 x

sysQa Almere ©

Quality Assurance in ICT / 11



Products of a review / inspection

- Defects
- Data
  - Time spend
  - # defect

Example: Inspection of a functional design

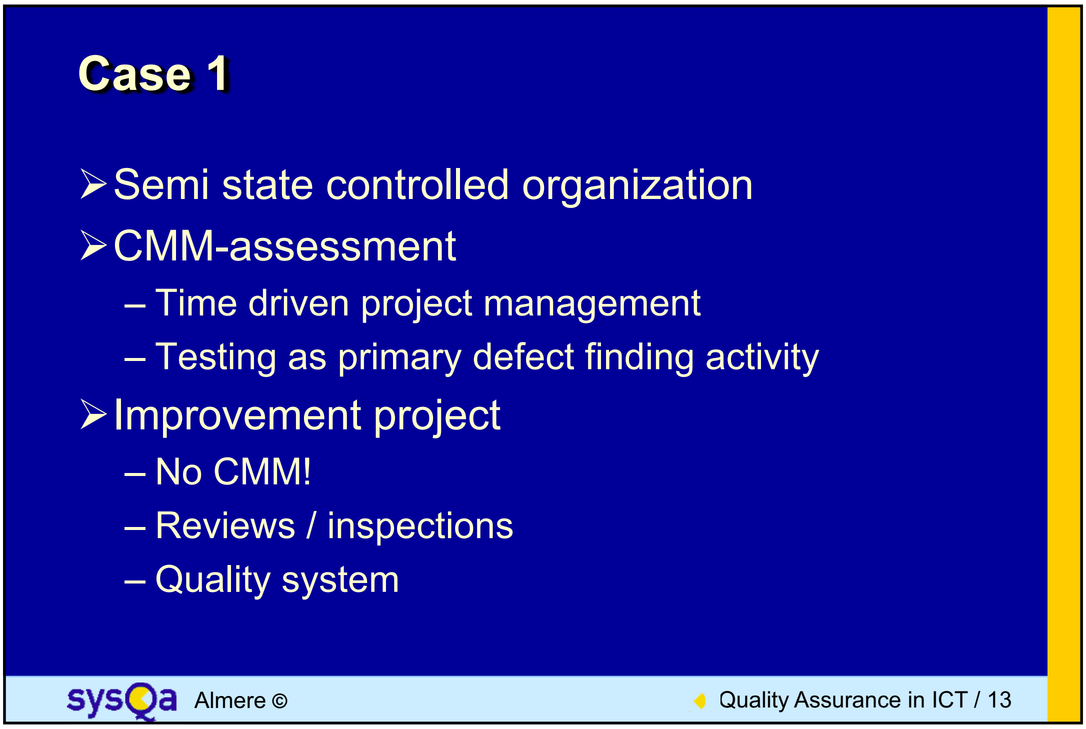
Time: 50 hours

12 mayor defects

ROI: (12*32)/50 = 7.68

sysQa Almere ©

Quality Assurance in ICT / 12



Case 1

- Semi state controlled organization
- CMM-assessment
  - Time driven project management
  - Testing as primary defect finding activity
- Improvement project
  - No CMM!
  - Reviews / inspections
  - Quality system

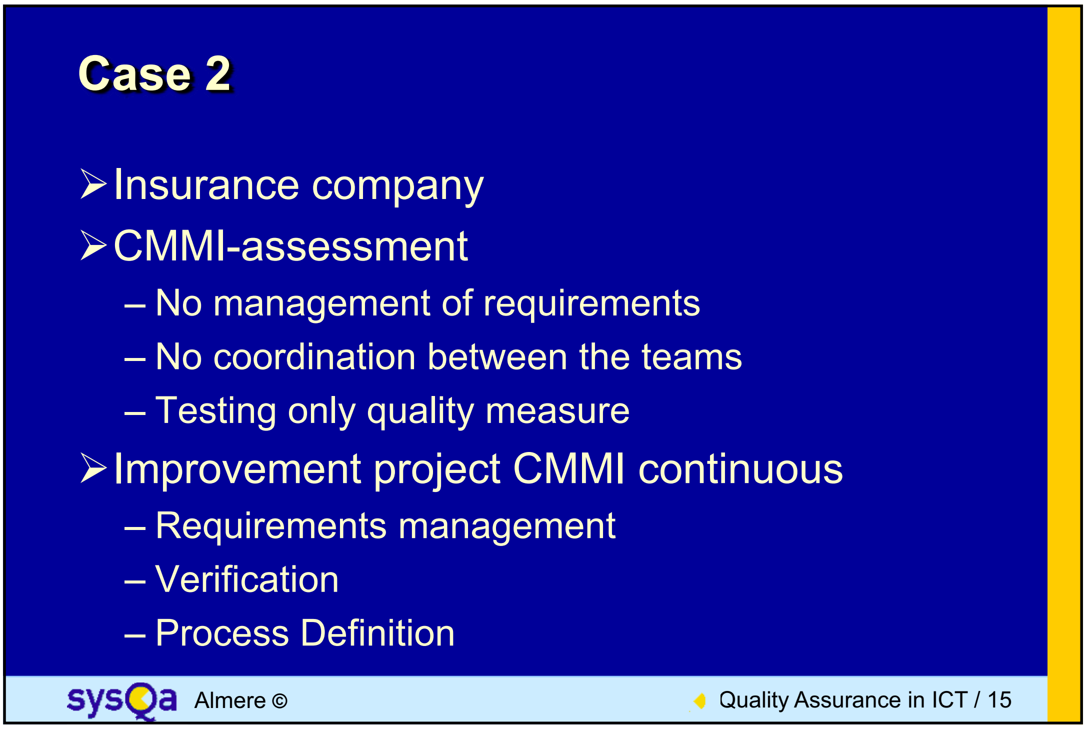
sysQa Almere © Quality Assurance in ICT / 13

Case 1 - results

- Implementation time: four months
- ROI first quarter: 7.92
- ROI second quarter: 6.26
- Investment in SPI earned back within 6 months
- Quality increased from 42% to 75%

Organization convinced

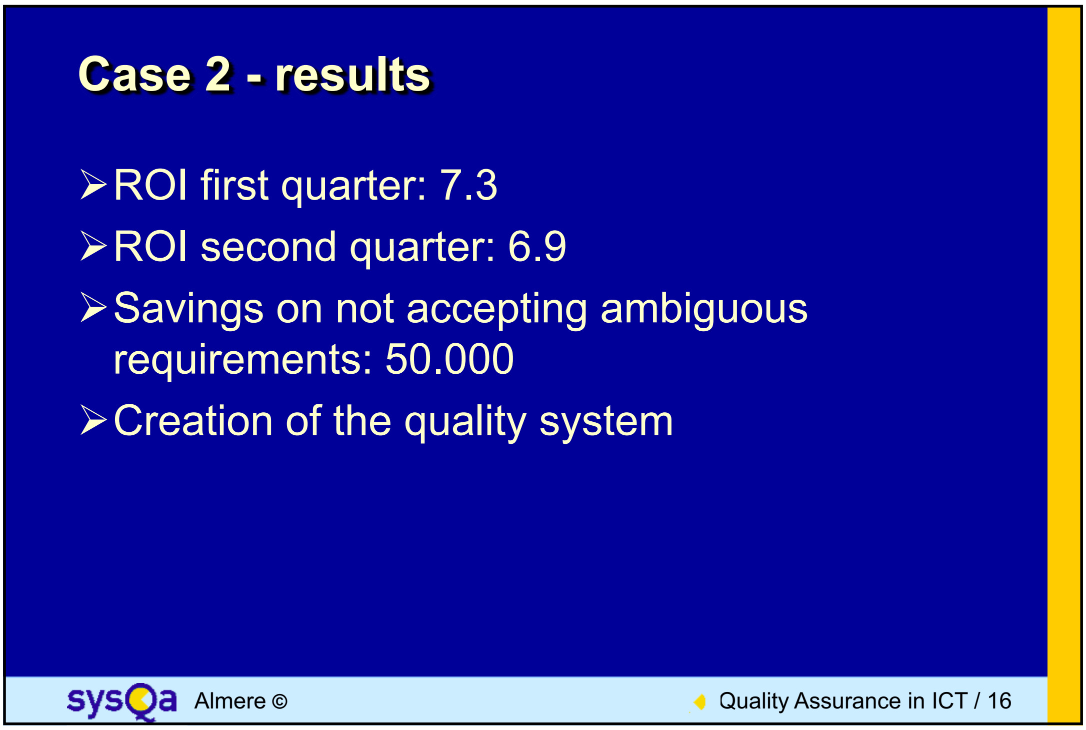
sysQa Almere © Quality Assurance in ICT / 14



Case 2

- Insurance company
- CMMI-assessment
  - No management of requirements
  - No coordination between the teams
  - Testing only quality measure
- Improvement project CMMI continuous
  - Requirements management
  - Verification
  - Process Definition

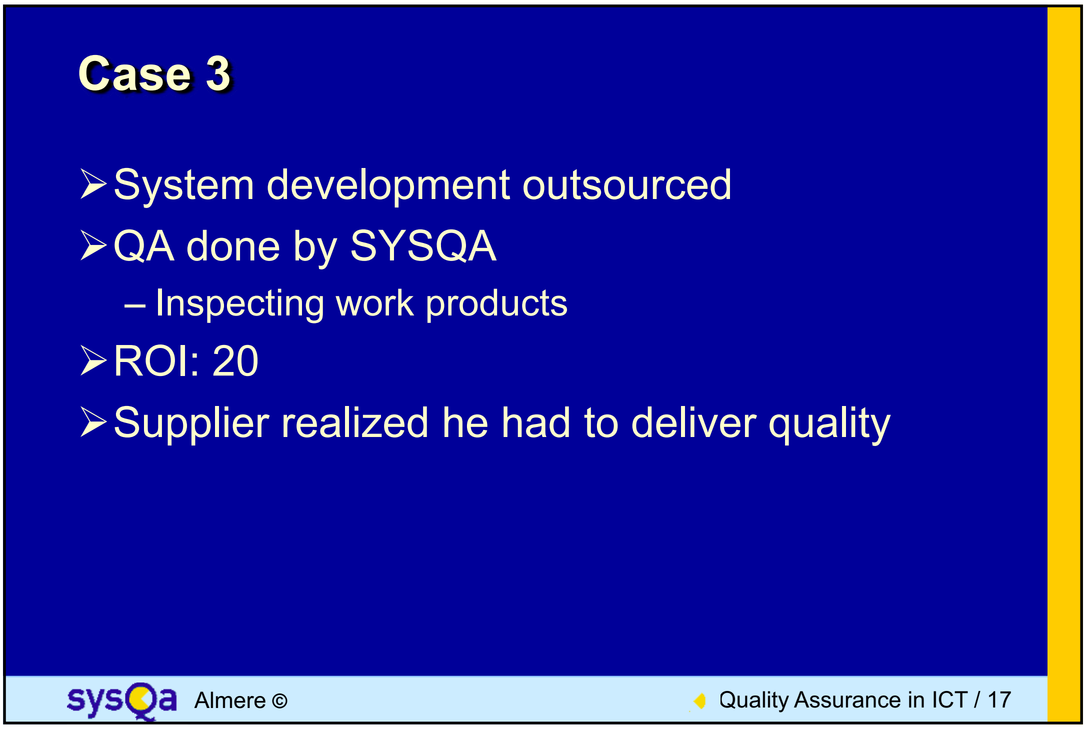
sysQa Almere © Quality Assurance in ICT / 15



Case 2 - results

- ROI first quarter: 7.3
- ROI second quarter: 6.9
- Savings on not accepting ambiguous requirements: 50.000
- Creation of the quality system

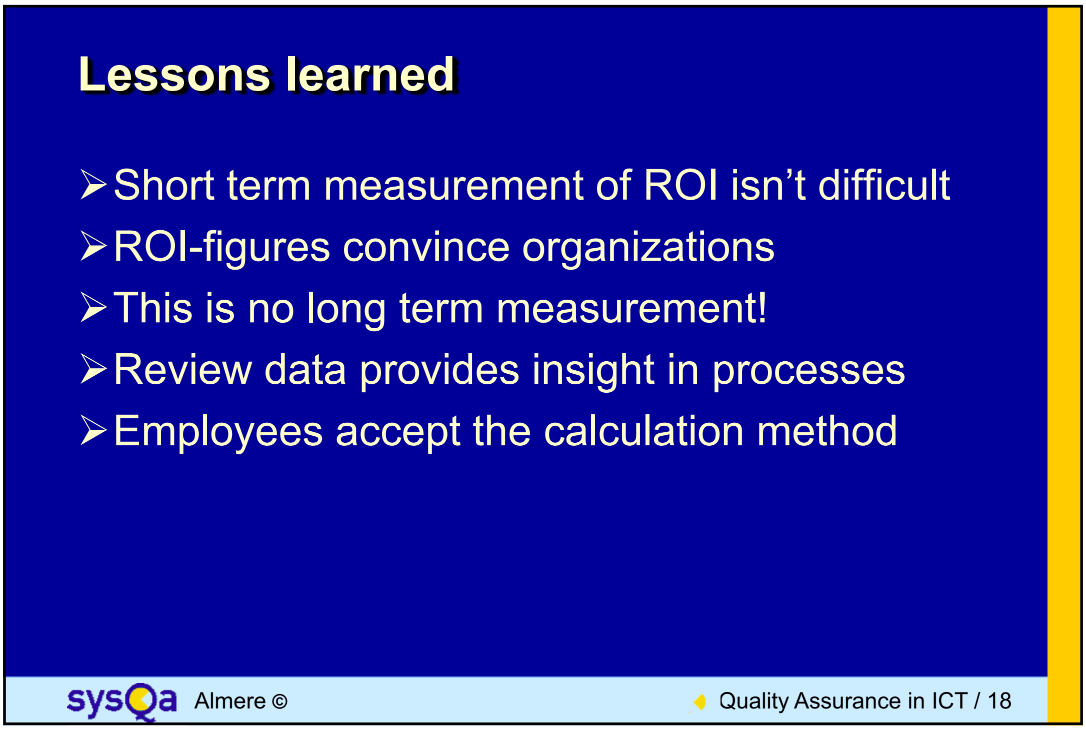
sysQa Almere © Quality Assurance in ICT / 16



Case 3

- ➢ System development outsourced
- ➢ QA done by SYSQA
  - – Inspecting work products
- ➢ ROI: 20
- ➢ Supplier realized he had to deliver quality

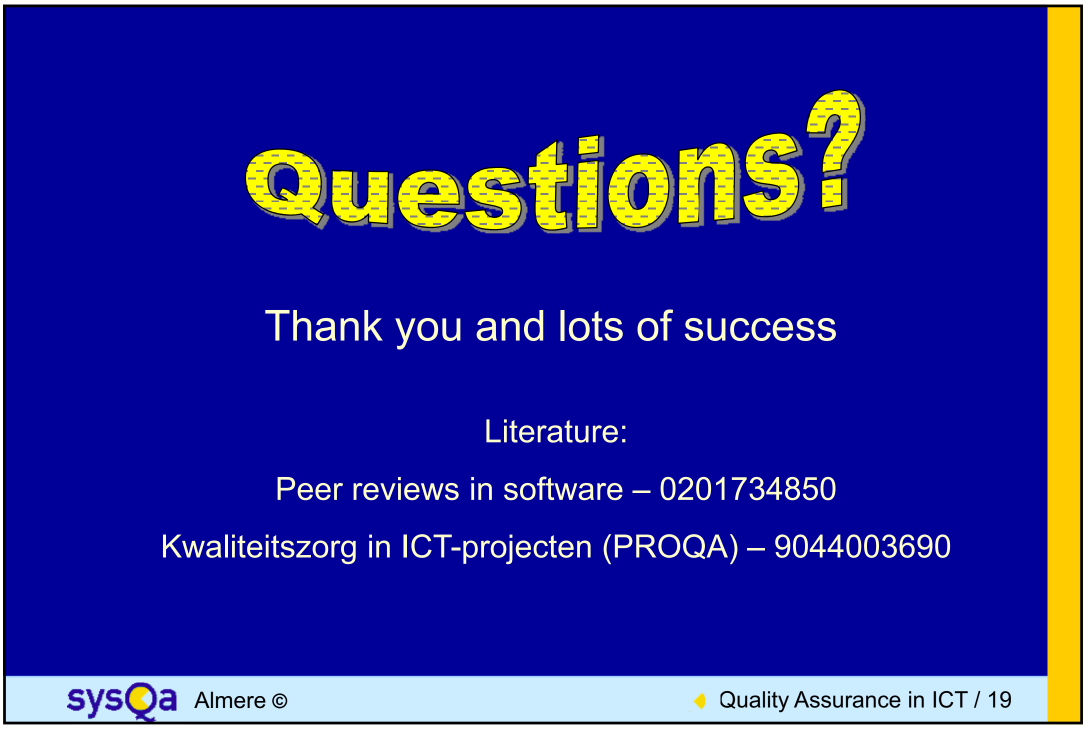
sysQa Almere © Quality Assurance in ICT / 17



Lessons learned

- ➢ Short term measurement of ROI isn’t difficult
- ➢ ROI-figures convince organizations
- ➢ This is no long term measurement!
- ➢ Review data provides insight in processes
- ➢ Employees accept the calculation method

sysQa Almere © Quality Assurance in ICT / 18



Questions?

Thank you and lots of success

Literature:

Peer reviews in software – 0201734850

Kwaliteitszorg in ICT-projecten (PROQA) – 9044003690

sysQa Almere ©

Quality Assurance in ICT / 19

CORRELATION BETWEEN CODING STANDARDS COMPLIANCE AND SOFTWARE QUALITY

Wojciech Basalaj

Programming Research, 9-11 Queens Road, Hersham, Surrey KT12 5LU, UK
Wojciech_Basalaj@programmingresearch.com

ABSTRACT

Software Quality has different meaning to different people. The ISO 9126 standard was developed to introduce clarity and establish a framework for quality to be measured. This paper aims to explore how *Internal Quality* characteristics of a software system (source code) can be measured effectively. Instead of relying on traditional software metrics, which are shown to be a poor predictor of underlying software quality, we advocate measuring compliance to a coding standard. We show qualitative and quantitative evidence of how adoption of a coding standard helps organizations in improving the quality of their C/C++ software.

Keywords: Software Quality Modelling, Coding Standards, Software Metrics, Statistical Analysis

1. ISO 9126 QUALITY MODEL

The ISO 9126-1 standard [4] has been introduced to formalise the notion of Quality of a Software System. 3 distinct aspects are considered:

- Internal Quality measured for a non-executable form of the Software System, e.g. its source code.
- External Quality, which pertains to the run-time behaviour of the system, as experienced during dynamic test.
- Quality in use, which addresses the degree to which user goals and requirements are fulfilled.

Internal and External Quality can be further categorised into 6 separate characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Each of these 6 characteristics can be further subdivided, and there are 27 sub-characteristics in total.

Quality in Use has been divided into 4 characteristics:

- Effectiveness
- Productivity
- Safety
- Satisfaction

ISO 9126-1 advocates measuring each of these characteristics, but does not specify how. Examples of suitable metrics are given in Technical Reports: 9126-2 [5], 9126-3 [6], 9126-4 [7]. The standard stipulates that with suitable choices of metrics Internal Quality should predict, or in other words correlate with External Quality, which in turn should predict Quality in Use.

In this study we will be focusing on the Satisfaction Quality in Use characteristic. We will attempt to demonstrate that this characteristic can indeed be predicted by measuring Internal Quality of a software system, see Section 4.1. We will also be examining empirical evidence of a correlation between Internal and External Quality measures, see Section 3.

Prior to conducting such a study we needed to settle on suitable metrics for Internal Quality. ISO 9126-3 [6] is a Technical Report that proposes such metrics. The vast majority of them are of the following form: percentage of items (functions, variables, etc.) meeting a specific requirement. There are a number of problems with such a definition of metrics. Their calculation cannot be easily automated, and their value needs to be determined by comparing implementation and design documents with specification. These metrics indicate how much work on the project has been completed, rather than the underlying quality of the implementation. Such metrics represent good project management practice for green-field projects, and cannot be applied easily when part of the system is re-engineered. Lastly, quality or lack thereof is not seen as an attribute of source code, as none of the proposed metrics are based on direct measurements on source code. This is against the guidance of ISO 9126-1 [4] page 15.

Prior to ISO 9126 there has been a vast amount of research devoted to software metrics [2]. These traditional metrics, such as Cyclomatic Complexity or Estimated Static Path Count, are concerned with the structure of a function, vocabulary of a source file, etc. Therefore, they may yield the same values for drastically different versions or stages of a Software Product, e.g. Cyclomatic Complexity for pseudo code stage may be the same as for the final implementation. Moreover, there is no well-defined and substantiated mapping for these metrics to ISO 9126 characteristics. We examine possible correlations of such software metrics with Quality in Use metrics in Section 4.2.

2. CODING STANDARDS

Nowadays, increasingly more emphasis is given to following best practice, and defining and enforcing coding standards, especially for high cost of failure software projects. Compliance to a coding standard is often treated as a pass/fail test. However, a different approach is possible, where the level of compliance is measured, either as the absolute number of violations for a particular source file, module or component, or normalised by the size of the entity, e.g. number of lines of code. This would allow correlating compliance with measurements of other aspects of the product, e.g. run time behaviour or user experience.

The most popular coding standard in the public domain for the C language is MISRA-C [12][13]. It constitutes a subset of the C language that restricts usage of poorly defined or unsafe constructs. Less emphasis is given to presentational aspects: naming conventions and layout. Until recently, no such definitive coding standard was available for the C++ language. The first and probably most complete is High Integrity C++ [14]. More recently, the Joint Strike Fighter Air Vehicle C++ Coding Standards [10] were released, demonstrating the growing industrial acceptance of using coding standards. Other C++ guidelines tend to focus on specific programming aspects [3][11][16][17].

The rules of these coding standards represent common pitfalls with developing in the corresponding programming language, and have been derived either from experience or on theoretical grounds, by examining the language specification [8][9]. Therefore, counting the number of violations of such rules in a Software Product appears well founded, and intuitively corresponds to a measure of its Internal Quality. This proposition is rigorously evaluated in Section 4.1.

3. QUALITATIVE RESULTS

We wanted to verify the proposal for measuring Internal Quality of a software product with real-world examples. We have engaged with some software companies, to find out what tangible benefits enforcement of a coding standard has given them. Two of them were able to offer broad qualitative statements, and these are documented in Section 3.1 and 3.2. However, they could not provide, in time for publication of this paper, any numerical data that would allow us to compare, for example, faults found in the field and compliance to a coding standard of specific software modules. However, another company had such data available, and we worked together to establish whether there were any correlations, see Section 4 for details.

3.1 Company A

They have been using MISRA-C:1998 [12] ever since historical process data have been collected. Some extra rules are enforced to do with naming conventions and limiting undefined behaviour. Typically, approximately 90% of the rules of this combined coding standard are adhered to for a project.

For a number of specific projects porting from one platform to another was required, and this was achieved with hardly any re-coding. This result was attributed to restricting undefined and implementation defined behaviour in their coding standard.

In their development process, unit testing occurs on a parallel track to coding, review and bench testing. By examining process data it was found that all the faults found in unit testing were also identified in the development track during code review (of which coding standard compliance is a part) or bench testing stages. Therefore, unit testing, despite being part of industry best practice, did not yield any new issues, apart from fulfilling its secondary role of verifying the specification. Subsequently, for some projects unit testing has been limited or dropped altogether in preference to proceeding straight to integration/system test.

3.2 Company B

The AUTOSAR[1] subset of MISRA-C:1998 [12] is used, as well as other proprietary coding standards, depending on the project, and this is mandated contractually.

The software projects are large, typically around 500KLOC. By defining a software platform, and making it conform to stricter rules on limiting implementation defined behaviour, they were able to migrate from one compiler and micro controller combination to another in a matter of weeks. This result is similar to that of Company A, see section 3.1.

Reuse is very common across projects, and coding standard rules on layout and naming conventions were found to be helpful in this regard.

4. QUANTITATIVE RESULTS

Company C has an ongoing programme for improving customer satisfaction. To this end they are collecting software fault reports from the field, and tracking them on a regular basis. The incidence of critical software faults tends to vary across their products, and the intention is to identify measurements on source code, i.e. Internal Quality metrics, that would correlate with these fault data, i.e. Quality in Use: Satisfaction metric. Once such source code factors are identified, it will be possible to re-engineer the software to minimise their value; and thus, likely to minimise the incidence of critical faults in released software.

Together with Company C we have collected code metrics for a number of their software products, and correlated them with the corresponding critical fault data. These code metrics fall into two categories:

- incidence of coding standard violations,
- traditional software metrics [2].

The results are documented in Sections 4.1 and 4.2 respectively.

4.1 Message Correlation

As a pilot study we focused on 18 software products written in C++, and owned by a single business unit. Critical fault data for each of the products was available, covering a period of 12 months. In order not to disadvantage large projects, we normalised these measurements of Quality in Use: Satisfaction by the size of the corresponding code base, i.e. amount of KLOC.

Rather than narrowing the study to some specific coding standard or guidelines (see Section 2), we decided to include as many coding rules as possible, in our search for the ones that will correlate with the fault data. QA C++, static analyser for C++ from Programming Research, includes nearly 900 rules ranging from ISO Compliance and Undefined Behaviour [9], Best Practice [3][11][14][16][17], to code layout conventions. This includes rules pertaining to individual source files as well as issues occurring across files, see Table 1 for examples.

confidence	msg#	QA C++ message text
99.5%	1512	'%1s' has external linkage and is declared in more than one file.
99%	1508	The typedef '%1s' is declared in more than one file.
99%	2085	For loop declaration of '%1s' is hiding existing declaration.
99%	4239	Class type control loop variable '%1s' modified in loop block.
97.5%	4217	Variable '%1s' is not accessed after this initialisation before it is next modified.
97.5%	4237	Class type control variable '%1s' not declared here.
97.5%	3600	This 'int' literal is an octal number.
95%	1505	The function '%1s' is only referenced in one translation unit.
95%	4243	Multiple class type loop control variables found: '%1s'.
95%	4325	Variable '%1s' is not accessed further.
95%	4004	Continue statement found.
95%	4208	Variable '%1s' is never used.
0%	2015	This function may be called with default arguments.

Table 1. Message correlation with critical fault data for a sample of QA C++ messages

For every software product we calculated the occurrence of each QA C++ message, and normalised the measurements by the size of the product in KLOC. While we could look for correlations between these raw measurements for fault data and message frequencies, this would make an unnecessary assumption that both of these populations of measurements were distributed similarly.

Instead, we decided to use ranks of the measurements only. If we were to order the software products according to fault data frequency, and for a given QA C++ message according to its frequency of occurrence, similarity between these two orderings would imply a positive correlation between the message and fault data. Considering that we are dealing with a large number of products, from statistical standpoint, it is not necessary that these orderings are identical, for there to be a significant correlation. Given that the number of permutations of 18 entities: 18! = 18*17*...*2 = 6,402,373,705,728,000 is a staggeringly large number, if a pair of orderings is within the 5% group that are the most similar, we can say with 95% confidence that they are correlated. 95% confidence interval is usually considered the minimum level to achieve statistical significance.

This leaves the question of how we are going to judge similarity between two given orderings of 18 products. Spearman's Rank Correlation Coefficient R_s [15] is a non-parametric statistical test, meaning that it works on the ranks of measurements. It evaluates to 1.0 if the orderings are exactly the same and -1.0 if they are exactly opposite, i.e. one is an inversion of the other sequence. The closer the value of R_s to 0 the less similar both orderings are. In this study we are only interested in positive correlations between Quality in Use and Internal Quality metrics: $R_s>0$. Given that we are dealing with 18 products, in order to have 95% confidence of a positive correlation between QA C++ message and fault data, the value of R_s needs to be no smaller than 0.401. Table 1 documents critical values of R_s for higher confidence intervals.

Confidence	95%	97.5%	99%	99.5%	99.9%
Critical Value of R_s	0.401	0.472	0.550	0.600	0.692

Table 2. Critical Values of Spearman's Rank Correlation Coefficient R_s for 18 entities

The first 12 rows of Table 1 list QA C++ messages that are positively correlated with critical fault data for the 18 software products under consideration, with at least 95% confidence. As an illustration the last row contains the message that has the value of R_s closest to 0. Figures 1-5 on page - 9 - display the correlation between the ranks of fault and message frequencies for each software product as a scatter plot, for a representative selection of messages from Table 1. Dots (software products) that lie on the $y=x$ (diagonal) line represent complete agreement between the ranks. In Figure 1 dots are much closer to the diagonal line than in Figure 5, which visually confirms the accuracy of the Spearman's Rank Correlation Coefficient. Figure 6 corresponds to the message with the smallest value of R_s; for convenience both positive $y=x$ and negative $y=19-x$ correlation lines are drawn. As can be seen dots are equally distant from both diagonal lines.

This result can be interpreted as follows: there is at least 95% likelihood that 12 QA C++ messages detailed in Table 1 are positively correlated with critical faults in 18 software products under consideration. This allows us to assume that by re-engineering these products to reduce the incidence of these messages, future occurrence of critical faults may also be reduced. As the organisation is interested in improving customer satisfaction, targeting these messages and monitoring their frequency can supplement the existing quality procedures.

It is worth pointing out that these 12 recommended messages are Best Practice rules, rather than rules targeting Undefined Behaviour, e.g. array access out of bounds, or division by 0. Such rules targeting potential 'bugs' are unlikely to occur frequently in the code. If for 18 products most frequencies are 0 apart from a few, the Spearman's Rank Correlation Coefficient will not exceed the critical value, and so the corresponding QA C++ message will not be flagged up as correlated with critical fault data. Therefore, it is necessary to supplement rules/messages identified by this statistical procedure with rules targeting bugs, portability issues, and other priorities identified for the software products in question.

4.2 Metrics Correlation

Apart from looking for correlations between critical faults and QA C++ messages, we were interested in examining whether traditional software metrics [2] could be of use. QA C++ calculates several function, file and class based metrics. We have recorded the average, maximum and standard deviation value of every metric for each of the 18 software products. We then calculated the values of Spearman's Rank Correlation Coefficient R_s between the critical fault and these metric data across the 18 products, which are collected in Table 3. Critical value of R_s at 95% confidence level is 0.401, and none of the metrics meet that for either average measurement, maximum or standard deviation. Therefore, we could not recommend any of these software metrics to be included in the quality initiative.

5. SUMMARY

In this paper we have proposed using coding standards compliance as a measure of Internal Quality of a Software System. The validity of this metric has been confirmed on a group of real-world software products, as for a number of coding rules it was found to correlate with a metric for Quality in Use: Satisfaction characteristic. Also, compliance to a coding standard has been found by two separate organisations to positively impact External Quality: Portability characteristic of their software.

User satisfaction is a concrete concept, and can be measured, e.g. by recording faults in released software. Coding standards compliance can also be easily measured, and subsequently improved, but does not directly map to improved user experience. However, this could be inferred, if a correlation between user satisfaction and compliance to coding rules is found, as is the case in this paper. An interesting topic for a future study would be to empirically demonstrate validity of this cause and effect hypothesis, by examining whether incidence of faults will be reduced in proportion to improvement in coding standards compliance.

Metric	avg	max	Std dev
Class metrics			
Coupling to other classes	0.041	0.005	0.043
Deepest inheritance	0.083	0.166	0.100
Lack of cohesion within class	-0.012	-0.061	-0.046
Number of methods declared in class	-0.098	-0.020	-0.023
Number of immediate children	0.055	0.025	0.061
Number of immediate parents	0.055	0.034	0.055
Response for class	-0.031	-0.057	-0.031
Weighted methods in class	-0.017	-0.034	-0.069
Function metrics			
Cyclomatic complexity	0.087	-0.141	-0.234
Number of GOTO's	-0.153	-0.238	-0.154
Number of code lines	-0.061	-0.068	-0.256
Deepest level of nesting	0.103	0.234	0.087
Number of parameters	0.129	0.192	0.122
Estimated static program paths	-0.362	n/a[§]	0.084
Number of function calls	-0.102	-0.019	-0.239
Number of executable lines	0.017	0.018	0.009
File metrics			
Comment to code ratio	0.283	0.153	0.287
Number of distinct operands	-0.220	-0.239	-0.304
Number of distinct operators	-0.035	0.260	0.124
Total preprocessed code lines	-0.074	0.142	-0.087
Total number of tokens used	-0.144	0.040	-0.138
Total unpreprocessed code lines	-0.073	0.077	-0.117
Total number of variables	-0.187	-0.044	-0.261

[§] For technical reasons we were not able to accurately calculate this value.

Table 3. Metrics correlation with fault data

Critical value of *Rs* at 95% confidence level is 0.401

REFERENCES

[1] Automotive Open System Architecture, www.autosar.org

[2] N.E. Fenton, S.L. Pfleeger. *Software Metrics: A Rigorous Approach*. 2nd edition. PWS, Boston, 1998

[3] M. Henricson, E. Nyquist, N. Erik. *Industrial Strength C++: Rules and Regulations*. Prentice Hall, 1997

[4] ISO/IEC 9126-1:2001. *Software engineering – Product quality – Part 1: Quality model.*

[5] ISO/IEC TR 9126-2:2003. *Software engineering – Product quality – Part 2: External metrics.*

[6] ISO/IEC TR 9126-3:2003. *Software engineering – Product quality – Part 3: Internal Metrics.*

[7] ISO/IEC TR 9126-4:2004. *Software engineering – Product quality – Part 4: Quality in use metrics.*

[8] ISO/IEC 9899:1990. *Programming languages – C.*

[9] ISO/IEC 14882:2003. *Programming languages – C++.*

[10] Lockheed Martin Corporation. *JSF AV C++ Coding Standards.* http://www.research.att.com/~bs/JSF-AV-rules.pdf 2005

[11] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 2nd edition. Addison Wesley, Boston, 2005

[12] MIRA, *MISRA-C:1998 - Guidelines for the Use of the C Language in Vehicle Based Software*. www.misra-c.com, 1998

[13] MIRA, *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. www.misra-c.com, 2004

[14] Programming Research. *High Integrity C++ Coding Standard Manual*. www.codingstandard.com, 2004

[15] S. Siegel. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill Book Company, Berkshire, 1956.

[16] H. Sutter, A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison Wesley, Boston, 2004

[17] H. Sutter. *Exceptional C++*. Addison Wesley, 1999

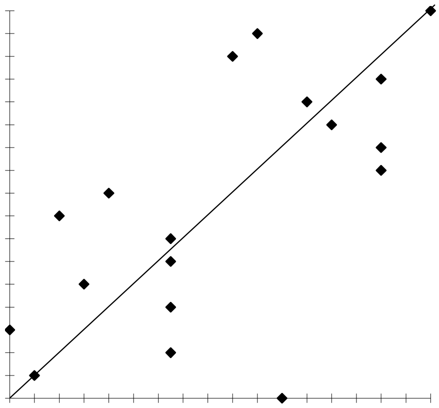


Figure 1. correlation for message 1512
R_s=0.649, confidence interval 99.5%

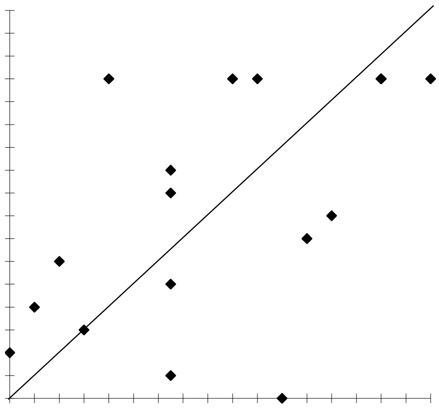


Figure 2. correlation for message 1508
R_s=0.568, confidence interval 99%

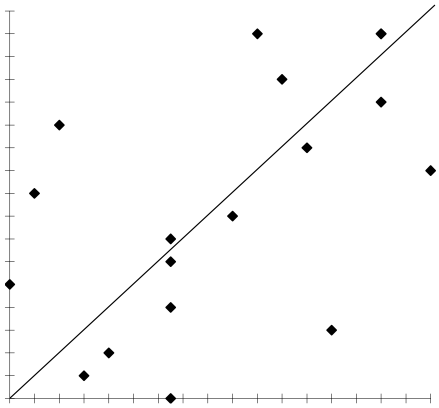


Figure 3. correlation for message 4217
R_s=0.533, confidence interval 97.5%

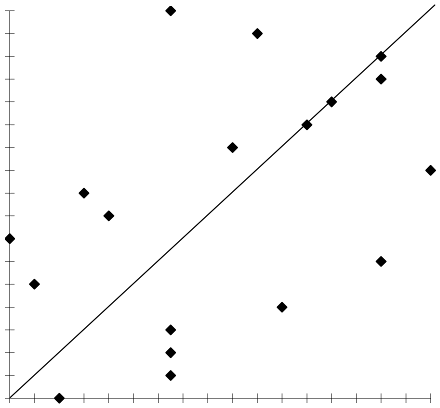


Figure 4. correlation for message 1505
R_s=0.466, confidence interval 95%

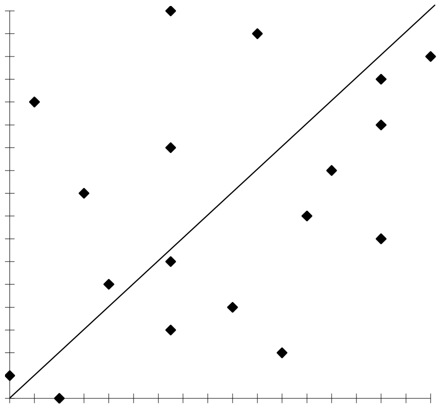


Figure 5. correlation for message 4208
R_s=0.403, confidence interval 95%

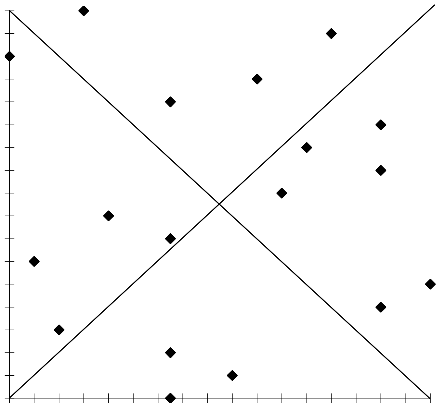


Figure 6. correlation for message 2015
R_s=0.001, i.e. no correlation

Verifying an implementation of SSH

Erik Poll[1⋆] and Aleksy Schubert[1,2⋆⋆]

[1] Radboud University Nijmegen, the Netherlands
[2] Warsaw University, Poland

Abstract. We present a case study in the formal verification of an open source Java implementation of SSH. We discuss the security flaws we found and fixed by means of formal specification and verification – using the specification language JML and the program verification tool ESC/Java2 – and by more basic manual code inspection. Of more general interest is the methodology we propose to formalise security protocols such as SSH using finite state machines. This provides a precise but accessible formal specification, that is not only useful for formal verification, but also for development, testing, and for clarification of official specification in natural language.

1 Introduction

The past decade has seen great progress in the field of formal analysis of security protocols. However, there has been little work or progress on verifying actual *implementations* of security protocols. Still, this is an important issue, because bugs can make an implementation of a secure protocol completely insecure. A fundamental challenge here is posed by the big gaps between (i) the official specification of a security protocol, typically in natural language; (ii) any models of (parts of) the protocol developed for formal verification of its security properties, e.g. using model checking; and (iii) actual implementations of the protocol. In an effort to bridge these gaps, we have performed a case study in the formal specification and verification of a Java implementation of SSH. We considered an existing implementation, MIDP-SSH[3], which is an actively maintained open source implementation for use on Java-enabled mobile phones. MIDP-SSH is a typical implementation in the sense that it is not written from scratch but based on an earlier one, re-using code from a variety of sources.

In order to express the properties to be verified for the source code, we used the Java Modeling Language (JML) [8]. JML is a specification language designed to describe properties of Java programs. It supports all the important features of the Java language e.g. inheritance, subtyping, exceptions etc. JML is supported by a range of tools for dynamic or static checking; for an overview see [2]. We used the extended static checker ESC/Java2 [3], the successor of ESC/Java [4]. This

[⋆] Supported by the Sixth Framework Programme of the EU under the MOBIUS project FP6-015905.

[⋆⋆] Supported supported by the Sixth Framework Programme of the EU under the SOJOURN project MEIF-CT-2005-024306.

[3] Available from `http://www.xk72.com/midpssh/`.

tool tries to verify automatically JML-annotated source code, using a weakest precondition calculus and an automated theorem prover.

The structure of the paper The informal code inspection is discussed in Section 2. The more formal analysis is discussed in Section 3. Section 1.1 below presents the methodology used to analyse the case study, and gives an overview of what we did in these two stages. We draw our conclusions and discuss possible future work in Section 4.

1.1 Methodology

After considering the security requirements of the application, our analysis of the implementation proceeded in several steps.

The first stage, described in Section 2, was an ad-hoc manual inspection of the source code. We familiarised ourselves with the design of the application, considered which parts of the code are security-sensitive, and looked for possible weaknesses. This led to discovery of some common mistakes – or at least bad practices which should be avoided in security-sensitive applications.

The next stages, described in Section 3, involved the use of the formal specification language JML and the program verification tool ESC/Java2. Here we can distinguish two stages:

- The second stage, discussed in Section 3.1, was the standard one when using ESC/Java2: we used the tool to verify that the implementation does not throw any runtime exceptions. For instance, the implementation might throw an ArrayIndexOutOfBoundsException due to incorrect handling of some malformed data packet it receives. This stage revealed some bugs in the implementation, where sanity checks on well-formedness of the data packets received were not properly carried out. This would only allow a DoS attack, by making the SSH client crash on such a malformed packet. Of course, for an implementation in a type-unsafe language such as C, as opposed to Java, these bugs would be much more serious, as potential sources of buffer overflow attacks.
  The process of using ESC/Java2 to verify that no runtime exceptions can occur, incl. the process of adding the JML annotations this requires, forces one to thoroughly inspect and understand the code. As a side effect of this we spotted a serious security weakness in the implementation, namely that it does not check the MAC of the incoming messages so it is vulnerable to certain replay attacks.
- The third stage, discussed in Section 3.2, was to verify that the Java code correctly implements the SSH protocol as officially specified in RFCs 4250-4254 [15, 13, 16, 14]. This required some formal specification of SSH. For this we developed our own formal specification of SSH, in the form of a finite state machine (FSM) which describes how the state of the protocol changes in response to the different messages it can receive. This is of course only a *partial* specification, as it specifies the *order* of messages but not their

precise format. Still, it turned out to be interesting enough, as we hope to demonstrate in this paper.

This third stage of the verification is probably the most interesting. Firstly, we found that obtaining the finite state machine from the natural language description in the RFCs was far from trivial, and it revealed some ambiguities and unclarities. It is not always clear what the response to an unexpected, unsupported or simply malformed message should be: some of these *may* or *should* be ignored but others *must* lead to disconnection.

Secondly, verifying that the implementation meets this partial specification as given by the FSM revealed some serious security flaws in the implementation. In particular, the implementation is vulnerable to a man-in-the-middle attack, where an attacker can request the username and password of the user *before* any authentication has taken place and before a session key has been established. A secure implementation should of course never handle such a request.

2 Stage 1: Informal, ad-hoc analysis

Prior to any systematic analysis of the application as discussed in the next section, we read the security analysis of the SSH protocol provided in the RFCs [15]. Then we extended the analysis to cover the issues closely related to the Java programming language and to the Java MIDP platform. We located the part of the source code which directly implements the protocol and tried to relate the results of the security analysis to the source code, but without trying to understand the logic of the implementation. In the course of these steps, we already spotted some (potential) security problems. Here is a description of the most important ones:

Weak/no authentication The SSH client does not store public key information for subsequent sessions: it will connect to any site and simply ask that site for its public keys, without checking this against earlier runs and asking the user to accept a new or changed public key. In other words, there is no real authentication before starting an SSH session. This is especially strange as the application stores certain session related information (i.e. host name, user name, and even password) in the MIDP permanent storage – record stores.

There is a countermeasure that allows the user to authenticate the server she or he is connecting to: the SSH client displays an MD5 hash of the server's public key as 'fingerprint' of the server it connects to. The user can check to see if this MD5 hash has the right value. Of course, the typical user will not check this.

Note that unauthenticated key exchange is a well-known and common security mistake; it is for instance listed in [6].

Poor use of Java access restrictions The implementation does not make optimal use of the possibilities that Java offers to restrict access to data, with the visibility modifiers (`public`, `private`, etc.) and the modifier `final` to make fields immutable.

For instance, the implementation creates an instance of the standard library class `java.lang.Random` for its random number generation. The reference to this object is stored in a *public* static field `rnd`. Untrusted code could simply modify this field, so that it for instance points to an instance of `java.lang.Random` with a known seed, or to an instance of some completely bogus subclass of `java.lang.Random` which does not produce random numbers at all. The field `rnd` should be `private` or `final` – or, better still, both – to avoid such tampering.

In all fairness, we should point out that for the current version of the MIDP platform the threat of some hostile application attacking the SSH client by changing its public fields does not seem feasible. A restriction of the MIDP platform is that at most one application – or *midlet*, as applications for the MIDP platform are called – is running at the same time, so a hostile application cannot be executing concurrently with the SSH midlet. Moreover, each time the SSH client is started it will initialise its fields from scratch. Still, such restrictions are likely to be loosened in the future, and the code of MIDP-SSH might be re-used in applications for other Java platforms where these restrictions do not apply.

A similar problem occurs with the storage of the contents of P- and S-boxes in the implementation. The class `Blowfish` in the implementation uses an array

```
final static int[] blowfish_sbox = { 0xd1310ba6, ... };
```

This integer array is final, so cannot be modified. However, the *content* of the array is still modifiable. The field is package-visible, which gives rather weak restrictions about who can modify is, as explained in [9], so hostile code could modify the S-boxes used by the SSH client, and at least create a DoS attack. The field should really be private and there is no reason why it cannot be. Again, for the MIDP platform this is not really a threat, due to its restrictions discussed above.

Checking if access modifiers can be tightened need not be done manually, but can be automated, for instance using JAMIT[4].

Control characters One of the security threats mentioned in the security analysis is the scenario when a malicious party sends a stream of control characters which erases certain messages to lure the user into performing an insecure action. Although the SSH client does interpret some control characters, there is no operation to ensure that only safe control sequences appear on the user's terminal.

Downloading of the session information The application implements functionality to download a description of an SSH session to execute. Such a description can contain the information about the user and a host name. The transfer of such information over the network in cleartext is an obvious compromise of the security as third parties can associate the login with the machine. Moreover,

[4] See `http://grothoff.org/christian/xtc/jamit/`

data downloaded in this way is not displayed to the user who demanded it. In this way it is easy to realize a spoofing attack which forwards the user to a fake SSH server which only steals the password.

3 Formal, systematic analysis using JML and ESC/Java2

The analysis using more formal methods consisted of two stages. The first stage was to verify that the implementation does not throw any runtime exceptions, e.g. due to null pointers, bad type casts, or accesses outside array bounds. The second one was to (partially) specify SSH, by means of a finite state machine, and verify that the implementation correctly implements this behaviour.

3.1 Stage 2: Exception Analysis

The standard first step in using ESC/Java2 is to check that the program does not produce any runtime exceptions. Indeed, often this is the only property one checks for the code. Although it is a relatively weak property, verifying it can reveal quite a number of bugs and exposes many implicit assumptions in the code. Just establishing exception freeness requires the formalisation of many properties about the code, as JML preconditions, invariants, and sometimes postconditions. For instance, invariants that certain reference fields cannot be null are needed to rule out NullPointerExceptions, and invariants that certain integer fields are not negative or have some maximum value are needed to rule out ArrayIndexOutOfBoundsExceptions.

Trying to check that no runtime exceptions occur with ESC/Java2 revealed some bugs in the implementation, namely missing sanity checks on the well-formedness of the data packets before these packets are processed. This means that the SSH client could crash with an ArrayIndexOutOfBoundsException when receiving some malformed packets. Such Denial-of-Service attacks are discussed in the RCFs.

The process of using ESC/Java2 to check that no runtime exceptions can occur – incl. the adding of all the JML annotations this requires – forces one to thoroughly inspect and understand the code. As a side effect of this we spotted a serious security weakness in the implementation, namely that it does not check the MAC of the incoming messages, so it is vulnerable to certain replay attacks.

The whole process of proving exception freeness, including fixing the code where required, took about two weeks.

3.2 Stage 3: Protocol specification and verification

In addition to just proving that the implementation does not throw runtime exceptions, we also wanted to verify that it is a correct implementation of the client side of the SSH protocol, as specified in the RFCs. But to do this, we first needed some formal specification of SSH.

Formal specification of SSH as FSM Unfortunately we could not find any formal description of SSH in the literature; the only formal description we could find [11] only deals with a part of the whole SSH protocol. Therefore we developed our own formal specification of SSH, in the form of a finite state machine (FSM) which describes how the state of the protocol changes in response to the different messages it can receive. This is of course only a *partial* specification, as it only specifies the *order* of messages but not their precise format. Still, this partial spec was interesting enough, as we hope to demonstrate in this paper.

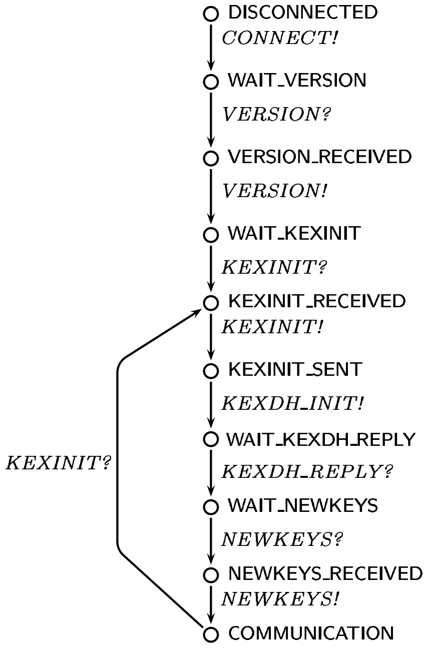


Fig. 1. A simplified view of the FSM specifying the behaviour of the SSH client, without optional features described in the RFCs that are not supported, and ignoring the aspects described in Fig. 2. The names of the transitions are the same names used in the RFCs. Labels ending with ! are outputs of the client to the server, labels ending with ? denote inputs to the client.

It turns out that the SSH protocol involves about 15 kinds of messages and its session key negotiation phase has about 20 different states. One complication in defining an FSM describing the client side behaviour of the protocol is that the SSH specifications present the protocol as a set of features which are partly obligatory and partly optional. A FSM that includes all these optional parts is given in Fig. 3 in the appendix. For simplicity, we focused our attention on those parts of the protocol that the implementation actually supports. This simplifies the overall behaviour to the FSM shown in Fig. 1, which corresponds to the left-most line of the states in the full specification given in the appendix.

Fig. 1 not only ignores options not implemented, but also includes an apparently common choice made in the implementation that is left open by the official specification. Section 4.2 of [16] states: "*When the connection has been established, both sides MUST send an identification string*". This specifies that both client and server must send an identification string, but does not specify the order in which they do this. In principle, it is possible for both sides to wait for the other to send the identification string first, leading to deadlock. The MIDP-SSH implementation chooses to let the client wait for an identification string from the server (the transition *VERSION?* in Fig. 1) before replying with an identification string (the transition *VERSION!* in Fig. 1). This appears to be the standard way of implementing this: OpenSSH makes the same choice. An earlier specification of SSH 1.5 [12, Overview of the Protocol] does prescribe this order; it is not clear to us why the newer specification [16] does not. Moreover, it is not clear if this is a deliberate underspecification or a mistake. Of course, one of the benefits of formalising specifications is that such issues come to light.

Fig. 1 does not tell the whole story, though. It only specifies the standard, correct sequence of messages, but does not specify how the client should react to unexpected, unsupported, or simply malformed messages. This is where much of the complication lies: some of these messages *may* or *should* be ignored, but others *must* lead to disconnection. Adding all the transitions for this to Fig. 1, (or, worse still, Fig. 3) would lead to a very complicated FSM that is hard to draw or understand, and very easy to get wrong. We therefore chose to specify these aspects in a separate FSM, given in Fig. 2.

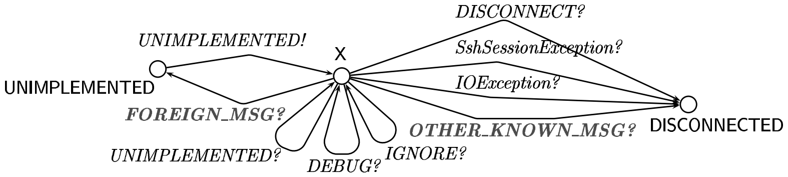


Fig. 2. Additional possible transitions from WAIT_KEXINIT onwards. X stands for any state from WAIT_KEXINIT onwards.

The SSH specification states that after the protocol version is negotiated, i.e. from the state WAIT_KEXINIT onwards, the client should always be able to handle a few messages in a generic way. Some of these messages should be completely ignored; some should lead to an *UNIMPLEMENTED!* reply, meaning the client does not support this message; some should lead to disconnection. This aspect is specified in a separate FSM: in the state WAIT_KEXINIT and any later state, the client should implement the additional transitions given in Fig. 2.

In Fig. 2 we use a few additional ad-hoc conventions to keep the diagram readable. *FOREIGN_MSGS?* stands for any message that is not explicitly known by the application. As noted above, all such messages should trigger the sending of the *UNIMPLEMENTED* message. Similarly, *OTHER_KNOWN_MSGS?* stands for any message that is known, but arrived in a wrong state – these messages lead to disconnection. This diagram is still a simplification because in some states certain known messages should be ignored rather than lead to disconnection, but we do not have space to discuss these details here.

Another ad-hoc convention are the labels *SshException?* and *IOException?*. These transitions represent two exceptional situations that can occur. Firstly, there is the possibility of an IO error (e.g. because the network or the server goes down), which is modelled by the *IOException?* transition. Secondly, there is the possibility that the incoming packet is of a known type but fails to meet the format specified in the RFCs (e.g. the value of the length field exceeds the size of the packet, or the MAC is incorrect), which is modelled by the *SshException?* transition. As you may have guessed, the names of these transitions are inspired the Java exceptions used in the implementation.

Discussion The finite state machines specifying SSH are implicit in the natural language specifications given in the RFCs, but were not so easy to extract, and highlighted some unclarities. We already mentioned the issue that description of the order of certain messages from client to server and back can be interpreted in several ways.

Whereas the names of various types of messages are well-standardised, and we use these in our diagrams, there is no explicit notion of state in the SSH specifications. So the names of the states in the diagrams are our invention. This lack of an explicit notion of state is a source of unclarity in the specification. In particular, [15, Sect. 9.3.5] asserts:

> If transmission errors or message manipulation occur, the connection is closed. The connection SHOULD be re-established if this occurs.

but it is hard to figure out which messages should be regarded as message manipulation at a given stage. The RFCs specify forbidden messages in several places, e.g. in [16, Sect. 7.1], e.g.

> Once a party has sent a SSH_MSG_KEXINIT message [. . .], until it has sent a SSH_MSG_NEWKEYS message (Section 7.3), it MUST NOT send any messages other than: [. . .]

but it is not obvious that messages other than those listed should be considered as 'manipulations' at this stage.

It would be better if the information about which messages are allowed, can be ignored, or must lead to disconnection in a given state is available in a more structured way. Now this information is spread out over several places in the

RFCs. An alternative to using FSMs might simply be a table of states and messages.

Another source of unclarity is the way the standard keywords are used in the specifications. There is an IETF standard which precisely defines the precise meaning of terms such as "MUST", "MAY", "RECOMMENDED", and "OPTIONAL" [1], but the SSH specification is not consistent in using these keywords. For example, [16, Section 4] says

> Key exchange *will* begin immediately after sending this identifier.

which presumably means that it "MUST" (and that any other behaviour "MUST" be considered as manipulation and lead to disconnection?).

Finally, in [16, Section 6] we noted that it is not clear if a well-formed packet may have a zero-length payload section or if such a packet should always be treated as malformed, because it is impossible to determine its type, which is crucial for any handling of the packet (the specification does not forbid such packets, but OpenSSH treats them as an error and quits the client).

3.3 Verification of MIDP-SSH

Before we even attempted a formal verification that the MIDP-SSH correctly implements the specification as given by the FSMs, it was easy to see that the implementation was not correct: it did not correctly record the protocol state, and accepted and processed many messages which following the FSMs should lead to disconnection. The prime example of this was that a request for username and password would be processed by the SSH client in any state.

Hence we improved the implementation before attempting formal verification: we re-factored the code so the handling of each message was done by a separate method, we improved the recording of the protocol state and added case distinctions based on the protocol state to obtain the right behaviour in each state.

To verify that the software correctly implemented the finite state machine, we then used AutoJML[5] [7], a tool that generates JML specifications (or Java code) from finite state machines.

This tool had to be adapted to cope with our use of several state diagrams to express various aspects of the behaviour, i.e. with Fig 2 expressing aspects of the behaviour that should be added to the overall behaviour in Fig. 1. (The alternative would have been to draw the very large finite state machine that would result from adding these aspects to the overall behaviour in Fig. 1.)

We added the specifications generated by AutoJML to the source code and verified them using ESC/Java2. This revealed there were still errors in the (already improved) implementation, where certain methods handled incoming messages in a different way than prescribed by the automata. Even though we were aware that the handling of exceptions is a delicate matter, and paid particular attention to this, we missed updates to the internal state variable in certain cases when the exceptions were thrown.

[5] Available from `http://autojml.sourceforge.net`

4 Conclusions

Now that there are various mature tools available to verify security properties of abstract security protocols, we believe it is time to tackle the next challenge, namely trying to verify the security of real implementations of such protocols.

This paper reports on an experiment to see if and how formal methods – in particular formal specification using finite state machines, the specification language JML, and the program checker ESC/Java2 – can be used for to verify an existing Java implementation of SSH. In the end, we managed to verify the implementation in the sense that it never throws an exception (which is maybe more a safety property than a security property) and that it correctly implements the SSH protocol as specified by finite state machines that we developed as formalisation of the official SSH specifications. Along the way we found and fixed several security flaws in the code. Some of these were found as a direct consequence of the verification, some were found more as a side-effect of having to thoroughly inspect and annotate the code to get it to verify. In this light, the method can also be regarded as a thorough expert analysis that results in a certificate which is closely related to the actual source code.

A general conclusion about our case study is that a formal specification of a security protocol in the form of a finite state machine is very useful. Given the complexity of real-life protocols, it is easy to get something wrong, as witnessed by the implementation we looked at. The specification of SSH as a finite state machine is formal, but still easy to understand by non-experts. We believe that providing such a description as part of official specification would be valuable, as it clarifies the specification and is also useful for development. Indeed, note that anyone who implements SSH will, as part of the work, have to implement a finite state machine that is described in the prose of the SSH RFCs and hence will re-do much of the work that we have done in coming up with the description of SSH as finite state machine.

The size of the SSH code we verified (just the code for the protocol, excluding the code for the GUI etc.) is around 4.5 kloc. The whole verification effort took about 6 weeks, including the time it took to understand and formalise the SSH specs, which was about 2 weeks. For widely used implementations of security protocols, say the implementation of SSL in the Java API, such an effort might be considered acceptable.

The second stage in our approach, ensuring the absence of runtime exceptions, can catch programming errors in the handling of individual messages, especially malformed ones. The third stage, verification of conformance to the FSM, can catch programming errors in the handling of sequences of messages, especially incorrect ones. Note that this complements conventional testing: testing – or, indeed, normal use of the application – is likely to reveal bugs in the handling of correctly formatted messages and correct sequences of such messages, but is less likely to reveal bugs in the handling of incorrectly formatted messages or incorrect sequences of messages, simply due to the limitless number of possibilities for this. So our approach may detect errors that are hard to find using testing.

A more practical issue is what the most convenient formalism or format for such finite state machines is, and which tools can be used to develop them. We developed our diagrams on paper and whiteboards, but with large number of arrows this becomes very cumbersome without some ad-hoc conventions and abbreviations. Maybe a purely graphical language is not the most convenient in the long run. Given the complexity of a real-life protocols, some way of separating different aspects in different finite state machines (as we have done with Fig. 1 describing the 'normal' scenario and Fig. 2 describing 'other' scenarios) seems important.

Related work An earlier paper [7] already investigated how a provably correct implementation could be obtained from an abstract security protocol for a very simple protocol. The AutoJML tool we used to produce JML specifications from the finite state machines can also produce a skeleton Java implementation. When developing an implementation for SSH from scratch, rather than examining an existing one as we did, this approach might be preferable. There are already efforts to generate code from abstract protocol descriptions, e.g. to generate Java code from security protocols described in the Spi calculus [10], or to refine abstract state machine (ASM) specifications to Java code [5].

Future work It would be interesting to repeat the experiment we have done for other implementations and for other protocols, i.e. trying to formalise other protocols using FSMs or other formalisms, and using these to check implementations. Of course, for an implementation that is not in Java, but say in C or C++, we might not have program checkers like ESC/Java2. Still, that a formalisation of a security protocol as a finite state machine, or in some other formalism, is also valuable for a human code inspection or for testing. Indeed, model-based testing could be used to test if an implementation of SSH conforms to our formal specification of the protocol.

In the end we only verified that the code correctly implements the protocol as described by the finite state machine, not that this protocol is secure, i.e. that it ensures authentication, integrity and confidentiality. Verifying that the full SSH protocol as described by our finite state machine from the appendix meets its security goals is still an interesting challenge to the security protocol verification community.

References

1. S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, The Internet Engineering Task Force, Network Working Group, March 1997.
2. L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
3. David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe et.al., editor, *CASSIS 2004*, number 3362 in LNCS. Springer, 2004.

4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245, New York, NY, USA, 2002. ACM Press.
5. Holger Grandy, Dominik Haneberg, Kurt Stenzel, and Wolfgang Reif. Developing provable secure m-commerce applications. In *Emerging Trends in Information and Communication Security*, volume 2995 of *LNCS*, pages 115–129, 2006.
6. Michael Howard, David leBlanc, and John Viega. *19 Deadly Sins of Software Security*. McGraw-Hill, 2005.
7. E.-M.G.M. Hubbers, M.D. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Security in Pervasive Computing, SPC'03*, volume 2802 of *LNCS*, pages 213–226. Springer-Verlag, 2004.
8. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Businesses and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer, 1999.
9. Gary McGraw and Ed Felten. *Securing Java*. Wiley, 1999. Available online at `www.securingjava.org`.
10. Benjamin Tobler and Andrew Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In E. Nardelli et.al., editor, *IFIP World Computer Congress - Certification and Security in Inter-Organizational E-Services (CSES)*, 2004.
11. David von Oheimb. Formal specification of the SSH transport layer protocol in HLPSL, 2004. Available online at `http://www.avispa-project.org/library/ssh-transport.html`.
12. T. Ylönen. The SSH (Secure Shell) Remote Login Protocol. Internet draft, The Internet Engineering Task Force, Network Working Group, NOV 1995. Available at `http://www.snailbook.com/docs/protocol-1.5.txt`.
13. T. Ylönen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, The Internet Engineering Task Force, Network Working Group, January 2006.
14. T. Ylönen. The Secure Shell (SSH) Connection Protocol. RFC 4254, The Internet Engineering Task Force, Network Working Group, January 2006.
15. T. Ylönen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, The Internet Engineering Task Force, Network Working Group, January 2006.
16. T. Ylönen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, The Internet Engineering Task Force, Network Working Group, January 2006.

A The full SSH protocol

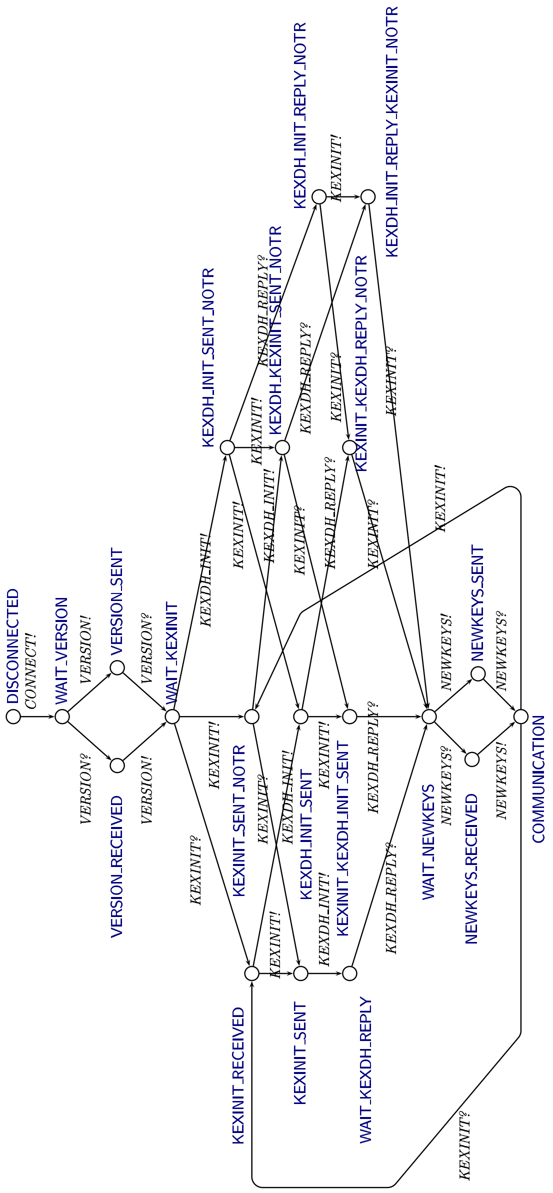
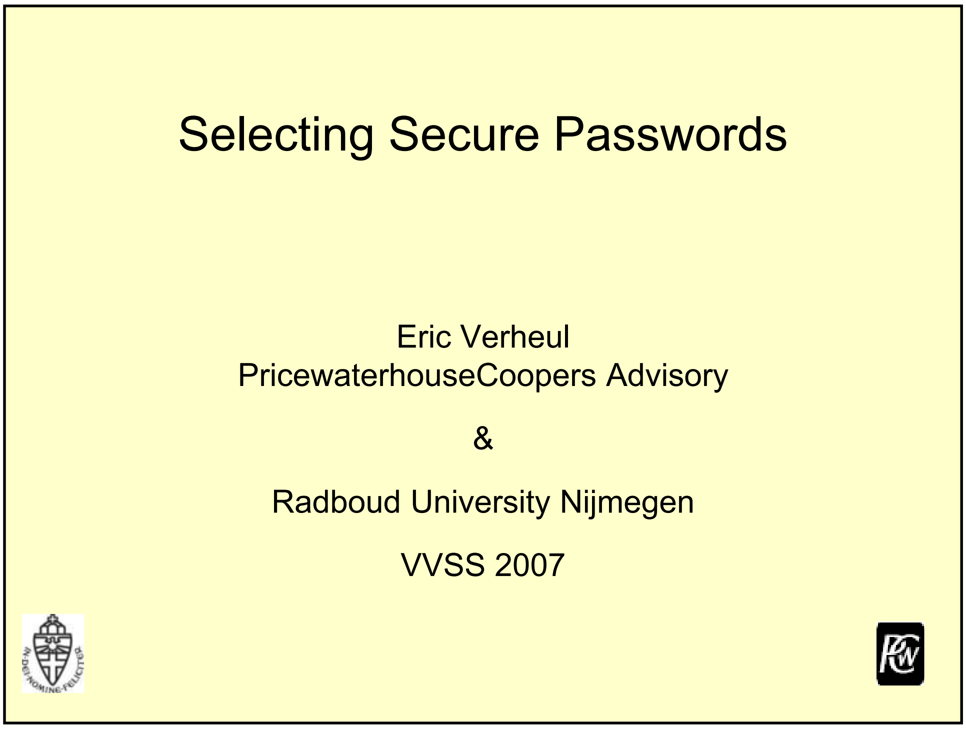


Fig. 3. Specification of the full client side behaviour of SSH, including all optional features. This diagram must still be extended with the additional aspects as given by the diagrams in Fig. 2



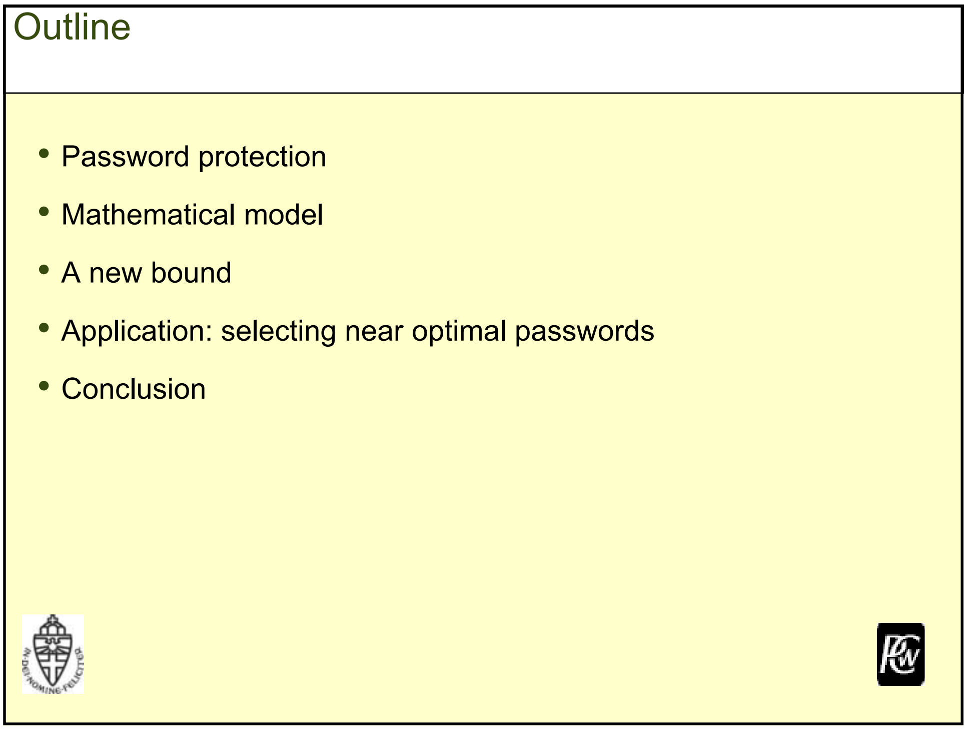
Selecting Secure Passwords

Eric Verheul
PricewaterhouseCoopers Advisory

&

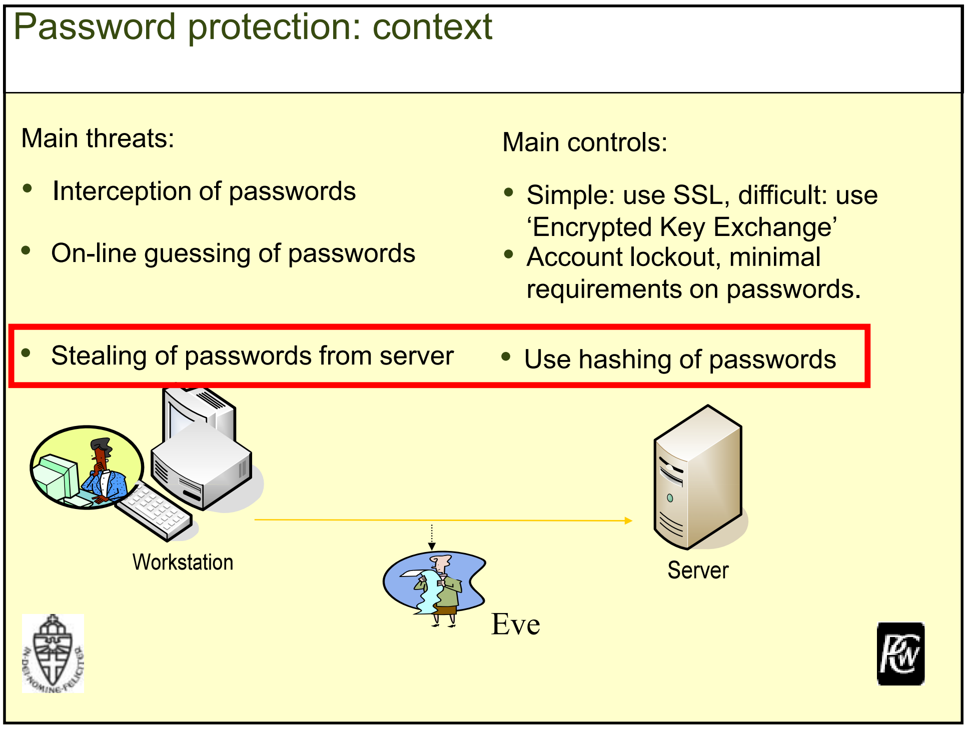
Radboud University Nijmegen

VVSS 2007



Outline

- Password protection
- Mathematical model
- A new bound
- Application: selecting near optimal passwords
- Conclusion



Password protection: context

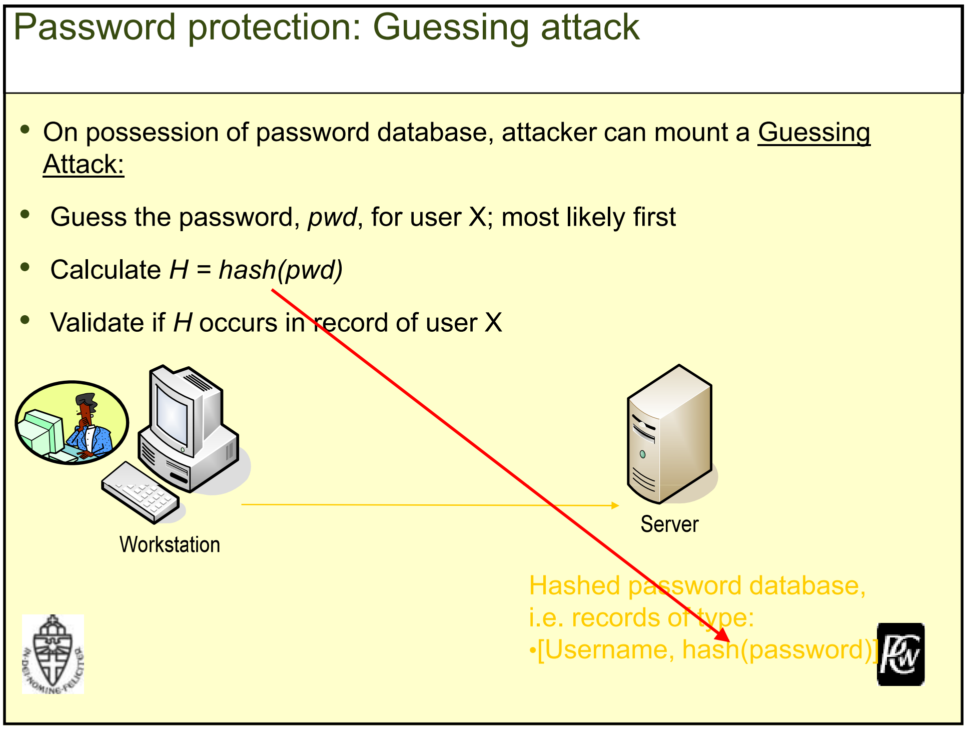
Main threats:

- Interception of passwords
- On-line guessing of passwords

Main controls:

- Simple: use SSL, difficult: use 'Encrypted Key Exchange'
- Account lockout, minimal requirements on passwords.

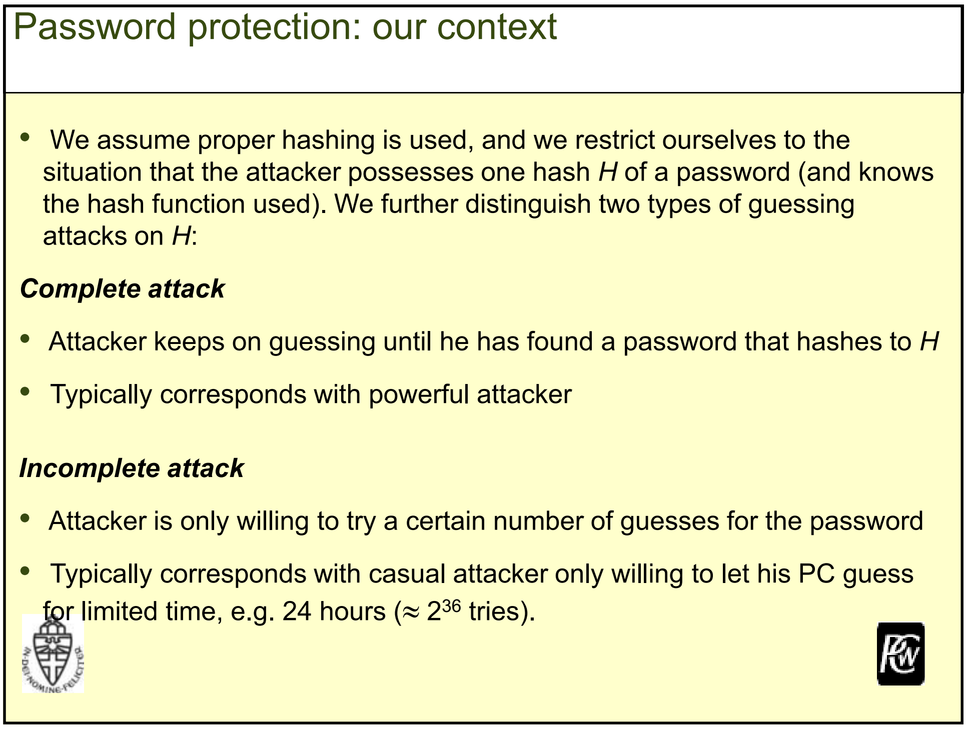
- Stealing of passwords from server
- Use hashing of passwords



Password protection: Guessing attack

- On possession of password database, attacker can mount a Guessing Attack:
- Guess the password, *pwd*, for user X; most likely first
- Calculate *H = hash(pwd)*
- Validate if *H* occurs in record of user X

Hashed password database,
i.e. records of type:
•[Username, hash(password)]



Password protection: our context

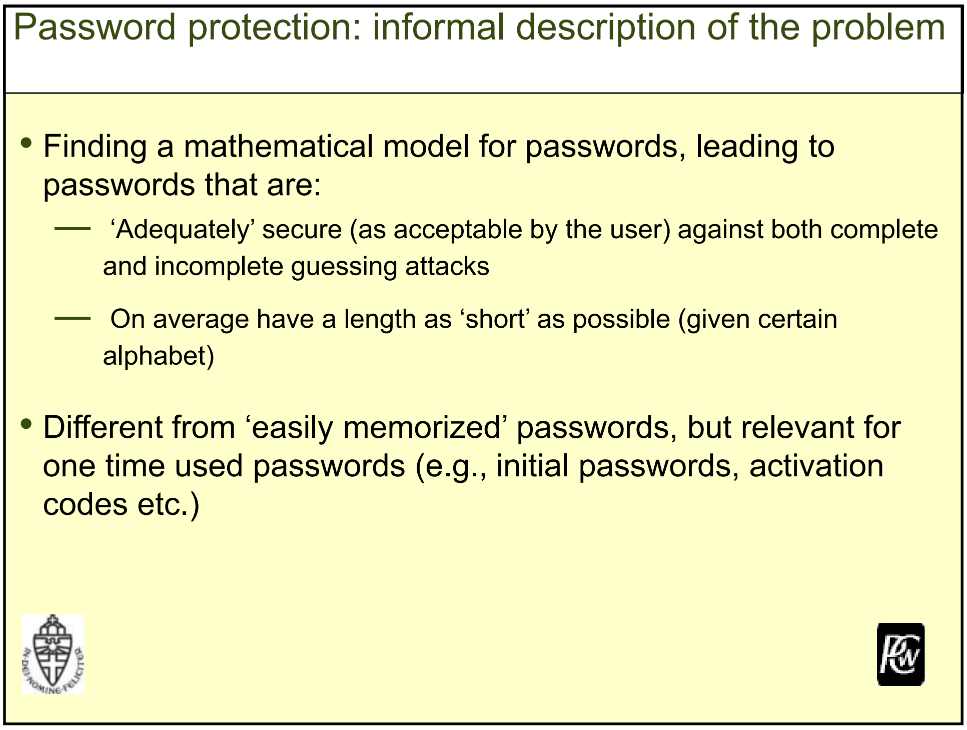
- We assume proper hashing is used, and we restrict ourselves to the situation that the attacker possesses one hash *H* of a password (and knows the hash function used). We further distinguish two types of guessing attacks on *H*:

Complete attack

- Attacker keeps on guessing until he has found a password that hashes to *H*
- Typically corresponds with powerful attacker

Incomplete attack

- Attacker is only willing to try a certain number of guesses for the password
- Typically corresponds with casual attacker only willing to let his PC guess for limited time, e.g. 24 hours (≈ 2^{36} tries).



Password protection: informal description of the problem

- Finding a mathematical model for passwords, leading to passwords that are:
  - ‘Adequately’ secure (as acceptable by the user) against both complete and incomplete guessing attacks
  - On average have a length as ‘short’ as possible (given certain alphabet)
- Different from ‘easily memorized’ passwords, but relevant for one time used passwords (e.g., initial passwords, activation codes etc.)

Mathematical model: representation of passwords

- Passwords correspond to a finite variable *X* with a discrete probability distribution (p_1 , p_2 , p_3 ,……, p_n) on *n* points (number of passwords), i.e. $p_i \geq 0$ and they sum up to one.
- We assume throughout that $\boxed{p_1 \geq p_2 \geq p_3 ,\ldots\ldots \geq p_n \geq 0.}$
- Evidently, $p_1 \geq 1/n$.

Mathematical model: measures of security

Guessing entropy: expected number of guesses in a *complete* off-line attack

$$\alpha = \sum_{i=1}^{n} i \cdot p_i$$

Min entropy: measure for resistance against an *incomplete* off-line attack

$$H_\infty = -\log_2(p_1)$$

Shannon entropy: measure for average length of passwords

$$H = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$$

$\boxed{\text{it simply follows that } H \geq H_\infty}$

Mathematical model: measures of security

Guessing entropy: expected number of guesses in a *complete* off-line attack

Example: Uniform Dist on n points

$$\alpha = \sum_{i=1}^{n} i \cdot p_i$$

(n+1)/2

Min entropy: measure for resistance against an *incomplete* off-line attack

$$H_\infty = -\log_2(p_1)$$

$\log_2(n)$

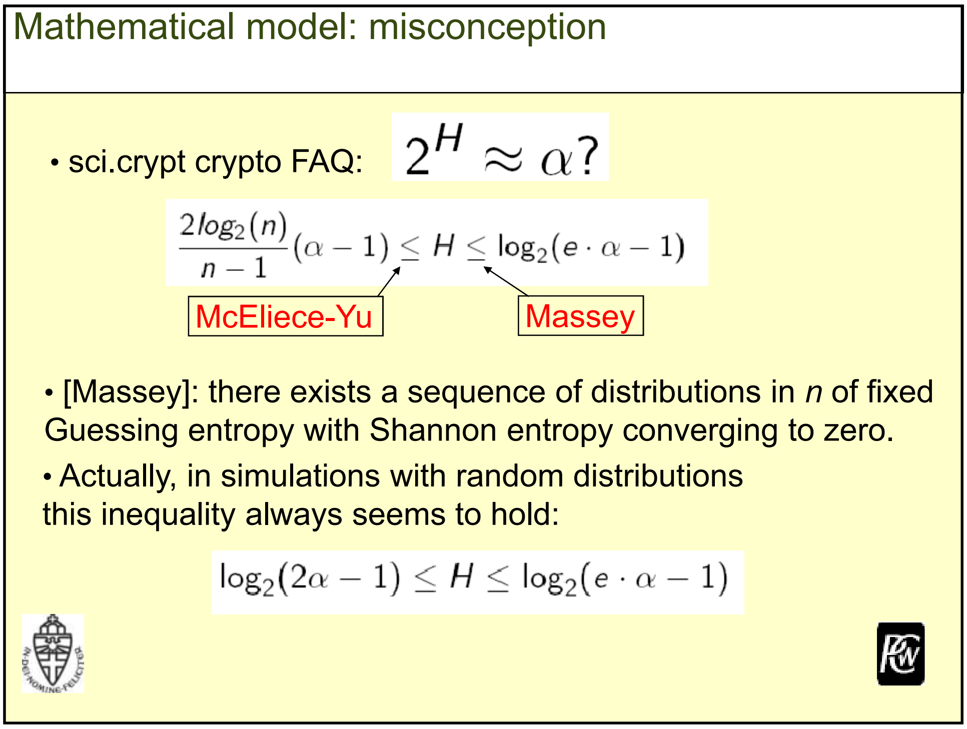
Shannon entropy: measure for average length of passwords

$$H = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$$

$\log_2(n)$

Mathematical model: problem formulation

- Given a value of guessing entropy α and a upper bound δ on p_1 (or equivalently a lower bound on the Min entropy): what is the minimal Shannon entropy possible?
- Efficiently find such distributions
- Efficiently generate such passwords
- Nist Special pub. 800-63: ‘electronic authentication guideline’ implicitly introduces this model, but does not pursues it.



Mathematical model: misconception

- sci.crypt crypto FAQ: $2^H \approx \alpha ?$

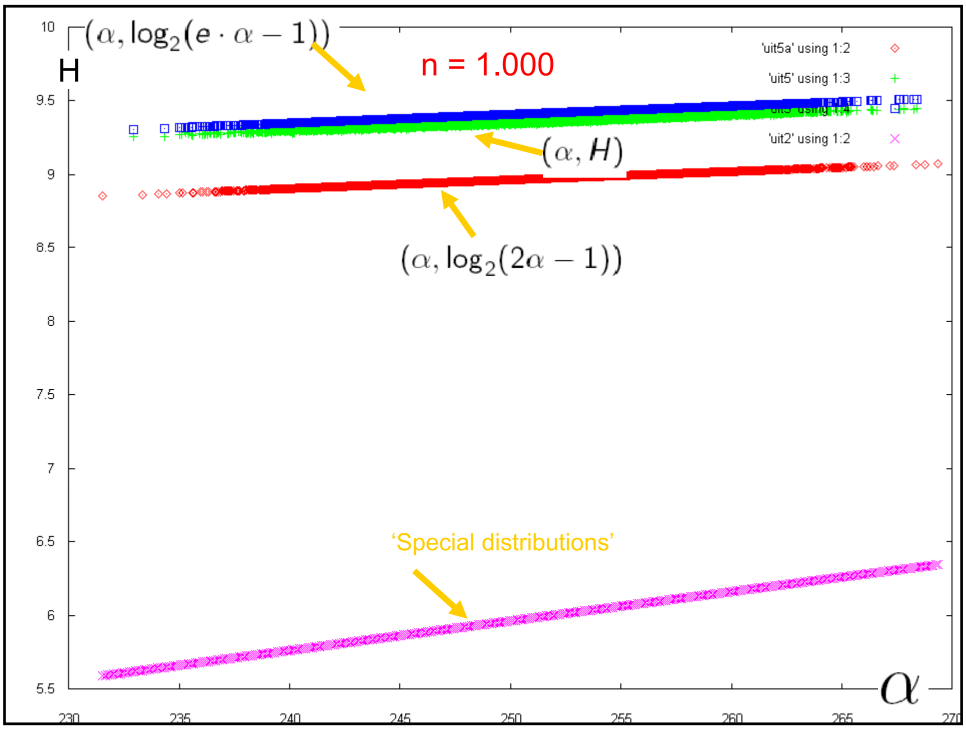
$$\frac{2 log_2(n)}{n-1}(\alpha - 1) \leq H \leq \log_2(e \cdot \alpha - 1)$$

McEliece-Yu

Massey

- [Massey]: there exists a sequence of distributions in *n* of fixed Guessing entropy with Shannon entropy converging to zero.
- Actually, in simulations with random distributions this inequality always seems to hold:

$$\log_2(2\alpha - 1) \leq H \leq \log_2(e \cdot \alpha - 1)$$



A new bound: relaxing the condition

- Looking for the minimal value of the Shannon entropy on $C_{n,\alpha}$

$$\{(p_1, \ldots, p_n) \in \mathbb{R}^n \mid \sum_{i=1}^{n} p_i = 1, \sum_{i=1}^{n} i \cdot p_i = \alpha, \boxed{\delta} \geq p_1 \geq p_2 \ldots \geq p_n \geq 0\}.$$

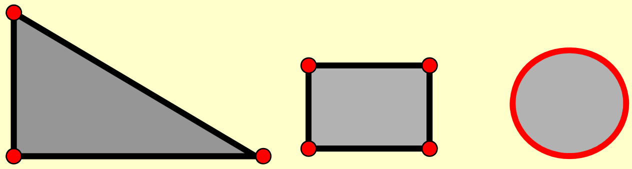
- First look for the minimal value the Shannon entropy H takes on the set $C_{n,\alpha}$

$$\{(p_1, \ldots, p_n) \in \mathbb{R}^n \mid \sum_{i=1} p_i = 1, \sum_{i=1} ip_i = \alpha, p_1 \geq p_2 \ldots \geq p_n \geq 0\}$$

- That is, the set of probability distributions on *n* points with a given guessing entropy α.

A new bound: extreme points convex sets

- $C_{n,\alpha}$ is closed convex set
- Every point is convex combination of extreme points



A new bound: extreme points of $C_{n,\alpha}$

The extreme points of $C_{n,\alpha}$ take the form $X_{j,k,n}$ for integers j, k satisfying $1 \leq j \leq 2\alpha - 1 \leq k \leq n$ and

$$\begin{array}{ccccccccccc} (& a_{j,k,n}, & a_{j,k,n}, & \cdots & a_{j,k,n}, & b_{j,k,n}, & \cdots & b_{j,k,n}, & 0, & \cdots & 0) \\ & \uparrow & & & \uparrow & \uparrow & & \uparrow & \uparrow & & \\ & 1, & 2, & \cdots & j, & j+1, & \cdots & k, & k+1, & \cdots & n, \end{array}$$

where

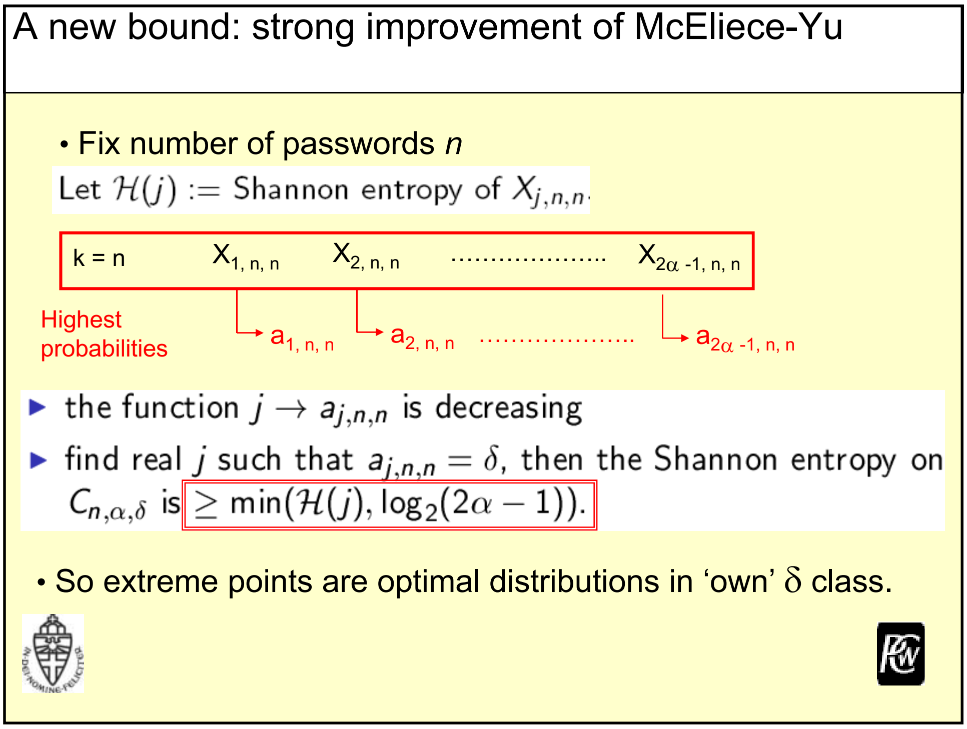
$$a_{j,k,n} = \frac{-2\alpha + 1 + j + k}{j \cdot k}; \, b_{j,k,n} = \frac{2\alpha - (j+1)}{k(k-j)},$$

and where we define $b_{j,k,n} = 1/(2\alpha - 1)$ for $j = 2\alpha - 1 = k$.

A new bound: extreme points of $C_{n,\alpha}$

$k = 2\alpha - 1$	$X_{1, 2\alpha -1, n}$	$X_{2, 2\alpha -1, n}$	…………………	$X_{2\alpha -1, 2\alpha -1,}$
$k = 2\alpha$	$X_{1, 2\alpha, n}$	$X_{2. 2\alpha. n}$	…………………	$X_{2\alpha -1, 2\alpha, n}$
$\vdots$				
$k = n$	$X_{1, n, n}$	$X_{2, n, n}$	…………………	$X_{2\alpha -1, n, n}$

These are vectors in R^n



A new bound: strong improvement of McEliece-Yu

- Fix number of passwords *n*

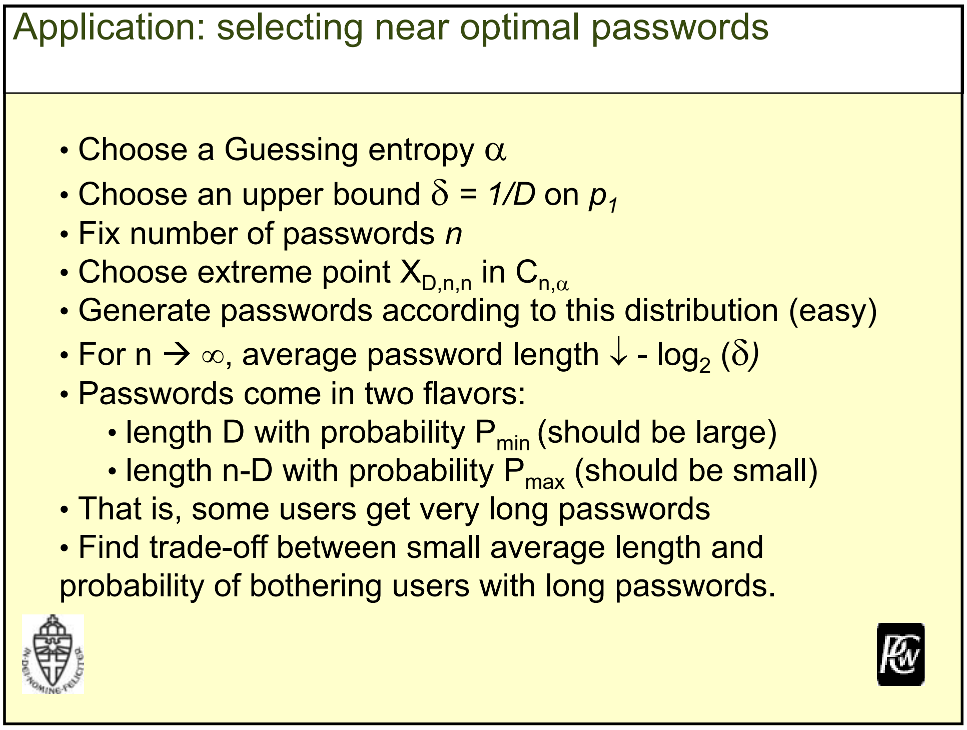
Let $\mathcal{H}(j) :=$ Shannon entropy of $X_{j,n,n}$.

k = n $X_{1, n, n}$ $X_{2, n, n}$ ………………… $X_{2\alpha -1, n, n}$

Highest probabilities $a_{1, n, n}$ $a_{2, n, n}$ ………………… $a_{2\alpha -1, n, n}$

- ▶ the function $j \to a_{j,n,n}$ is decreasing
- ▶ find real j such that $a_{j,n,n} = \delta$, then the Shannon entropy on $C_{n,\alpha,\delta}$ is $\boxed{\geq \min(\mathcal{H}(j), \log_2(2\alpha - 1)).}$

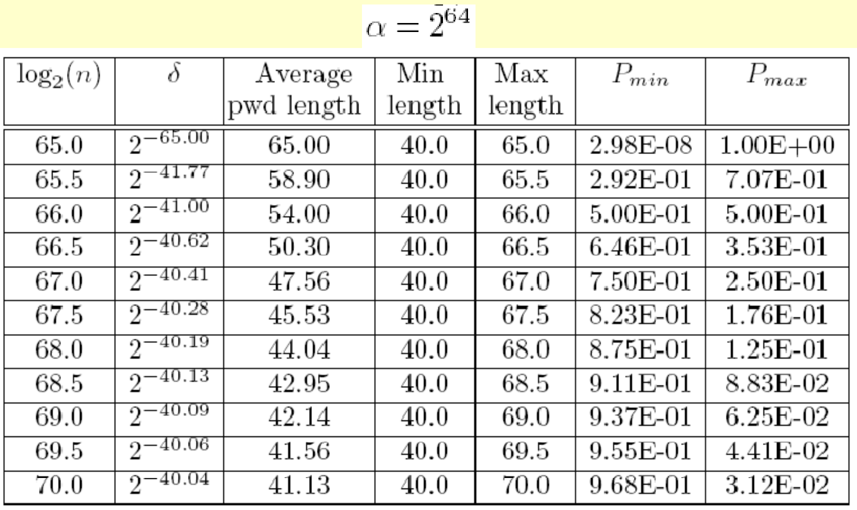
- So extreme points are optimal distributions in 'own' δ class.



Application: selecting near optimal passwords

- Choose a Guessing entropy α
- Choose an upper bound δ = *1/D* on p_1
- Fix number of passwords *n*
- Choose extreme point $X_{D,n,n}$ in $C_{n,\alpha}$
- Generate passwords according to this distribution (easy)
- For n → ∞, average password length ↓ - $\log_2$ (δ)
- Passwords come in two flavors:
  - length D with probability P_{min} (should be large)
  - length n-D with probability P_{max} (should be small)
- That is, some users get very long passwords
- Find trade-off between small average length and probability of bothering users with long passwords.

Application: selecting near optimal passwords



$\alpha = 2^{64}$

$\log_2(n)$	δ	Average pwd length	Min length	Max length	P_{min}	P_{max}
65.0	$2^{-65.00}$	65.00	40.0	65.0	2.98E-08	1.00E+00
65.5	$2^{-41.77}$	58.90	40.0	65.5	2.92E-01	7.07E-01
66.0	$2^{-41.00}$	54.00	40.0	66.0	5.00E-01	5.00E-01
66.5	$2^{-40.62}$	50.30	40.0	66.5	6.46E-01	3.53E-01
67.0	$2^{-40.41}$	47.56	40.0	67.0	7.50E-01	2.50E-01
67.5	$2^{-40.28}$	45.53	40.0	67.5	8.23E-01	1.76E-01
68.0	$2^{-40.19}$	44.04	40.0	68.0	8.75E-01	1.25E-01
68.5	$2^{-40.13}$	42.95	40.0	68.5	9.11E-01	8.83E-02
69.0	$2^{-40.09}$	42.14	40.0	69.0	9.37E-01	6.25E-02
69.5	$2^{-40.06}$	41.56	40.0	69.5	9.55E-01	4.41E-02
70.0	$2^{-40.04}$	41.13	40.0	70.0	9.68E-01	3.12E-02

Conclusion