Computer Safety, Reliability and Security

20th International Conference, SAFECOMP 2001
Budapest, Hungary, September 26-28, 2001
Proceedings
Preface

This year we celebrated another anniversary: after 20 years of SAFECOMP in 1999, this was the 20th SAFECOMP since its inauguration in 1979. This series of events focuses on critical computer applications. It is intended to be a platform for knowledge transfer between academia, industry, and research institutions. Papers are solicited on all aspects of computer systems in which safety, reliability, and security (applied to safety in terms of integrity and availability) are of importance.


Authors from 13 countries responded to the Call for Papers, and 10 countries were represented in the final program. The proceedings include 20 papers plus 3 invited papers, covering the areas Reliability Assessment and Security, Safety Case and Safety Analysis, Testing, Formal Methods, Control Systems, and this year covering new grounds with a special emphasis on Human-Machine Interface, Components off the Shelf, and Medical Systems.

As Program Chair of SAFECOMP 2001 I would like to thank all the authors who answered our Call for Papers, the selected ones for providing their papers in time for the proceedings and presenting them at the conference, the members of the International Program Committee for the review work and guidance in preparing the program, the General Chair and the Organizing Committee for all the visible and invisible work while preparing the conference, the sponsors and the co-sponsors for their financial and non-material support, and also all those unnamed who helped with their effort and support to make SAFECOMP 2001 a fruitful event and a success.

I hope that all those who attended the conference gained additional insight and increased their knowledge, and that those reading this collection of articles after the event will be motivated to take part in the next SAFECOMP in Catania, Italy, in 2002.

July 2001

Udo Voges
Introduction Remarks from the Organizing Committee

Scientists and software and computer engineers are coming to the event of SAFECOMP 2001, the 20th Conference on Computer Safety, Reliability, and Security to be held in Budapest this year.

Issues and problems that are related to the safety, reliability, and security of computers, communication systems, components of the networked world have never been so much at the center of attention of system developers and users as today. The emerging world of the eEconomy is becoming more and more dependent on the availability of reliable data and information, control commands, and computing capacity that are used everywhere: in academia, research institutes, industry, services, businesses as well as the everyday activity of people. Huge material values, correct operation of critical systems, health and life of people may depend on the availability and validity of data, correctness of control information, fidelity of the results of processing, as well as on the safe delivery of these data to the recipients.

It is not enough to tackle problems of individual computers or communication equipment alone. The complex web of networks connected and interrelated, the huge number of active processing entities that receive and produce data to this “world-wide-web” make the task of ensuring safe and secure operation far more complex than in isolated, stand alone systems or smaller local networks of computers. Moreover, considerations on the technological aspects of security are no longer sufficient. We have to work out effective methods as to how to investigate the behavior of the huge interconnected world of computers and communication systems together with their users and operators with very different tasks, work traditions, skills, and educational backgrounds.

This leads us to the question of not only computer safety, reliability, and security, but the safety, reliability, and security of the accumulated and transferred knowledge, i.e. knowledge management: knowledge acquisition, storage, transfer, processing, understanding, and evaluation.

When we use the term knowledge, we consider not only technical systems, but people, and their creativity and ability to use the data and information provided by technical means, computers, and networks. We agree with the statement of T.H. Davenport and L. Prusak, according to which knowledge is originated from working brains, not technical systems.

More and more countries and governments announce plans and strategies toward the establishment of an information society, eEconomy, etc, on all continents. One may notice that some of the most crucial points in these programs or strategies are trust, safety, confidence, and reliability of data.
Computers, informatics, data, and knowledge processing reshape our future, change the way we live, work, communicate with each other and spend our vacation. The future, and our success or failure, depend very much on the extent to which we can include, and attract as many people as possible (hopefully everybody) into the world offered by the Internet revolution, the world of the information society. Users are very much aware of the safety of the systems upon which their activity or their work depends. Hence their involvement is also very much dependent on their trust of and confidence in this new environment.

The conference attracts specialists working toward creating a safe environment.

The Organizing Committee, the community of informatics and “knowledge” specialists hosting the conference express their gratitude to all those – organizers, invited speakers, presenters and participants – who have worked for this event, sharing the results of their research and thus making the conference a fruitful meeting.
# Committees

## International Program Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Name</th>
<th>Country</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuart Anderson</td>
<td>GB</td>
<td>Floor Koornneef</td>
<td>NL</td>
<td></td>
</tr>
<tr>
<td>Helmut Bezecny</td>
<td>DE</td>
<td>Vic Maggioli</td>
<td>US</td>
<td></td>
</tr>
<tr>
<td>Robin Bloomfield</td>
<td>GB</td>
<td>Odd Nordland</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Andrea Bondavalli</td>
<td>IT</td>
<td>Alberto Pasquini</td>
<td>IT</td>
<td></td>
</tr>
<tr>
<td>Helmut Breitwieser</td>
<td>DE</td>
<td>Gerd Rabe</td>
<td>DE</td>
<td>(EWICS Chair)</td>
</tr>
<tr>
<td>Peter Daniel</td>
<td>GB</td>
<td>Felix Redmill</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>Bas de Mol</td>
<td>NL</td>
<td>Francesca Saglietti</td>
<td>DE</td>
<td></td>
</tr>
<tr>
<td>István Erényi</td>
<td>HU</td>
<td>Erwin Schoitsch</td>
<td>AT</td>
<td>(General Chair)</td>
</tr>
<tr>
<td>Robert Garnier</td>
<td>FR</td>
<td>Ian Smith</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>Robert Genser</td>
<td>AT</td>
<td>Meine van der Meulen</td>
<td>NL</td>
<td></td>
</tr>
<tr>
<td>Chris Goring</td>
<td>GB</td>
<td>Udo Voges</td>
<td>DE</td>
<td>(IPC Chair)</td>
</tr>
<tr>
<td>Janusz Gorski</td>
<td>PL</td>
<td>Marc Wilikens</td>
<td>IT</td>
<td></td>
</tr>
<tr>
<td>Erwin Großpietsch</td>
<td>DE</td>
<td>Rune Winther</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Maritta Heisel</td>
<td>DE</td>
<td>Stefan Wittmann</td>
<td>DE</td>
<td></td>
</tr>
<tr>
<td>Chris Johnson</td>
<td>GB</td>
<td>Janus Zalewski</td>
<td>US</td>
<td></td>
</tr>
<tr>
<td>Mohamed Kaaniche</td>
<td>FR</td>
<td>Zdislaw Zurakowski</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>Karama Kanoun</td>
<td>FR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Organizing Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>István Erényi</td>
<td>HU</td>
<td>(Local Chair)</td>
</tr>
<tr>
<td>Emese Kövér</td>
<td>HU</td>
<td></td>
</tr>
<tr>
<td>Erwin Schoitsch</td>
<td>AT</td>
<td></td>
</tr>
<tr>
<td>Mária Tóth</td>
<td>HU</td>
<td></td>
</tr>
</tbody>
</table>

## External Reviewers

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marc Mersiol</td>
<td>FR</td>
</tr>
<tr>
<td>Thomas Ringler</td>
<td>DE</td>
</tr>
<tr>
<td>Mark A. Sujan</td>
<td>DE</td>
</tr>
<tr>
<td>Helene Waeselynck</td>
<td>FR</td>
</tr>
</tbody>
</table>
List of Contributors

Jean Arlat
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4
France
arlat@laas.fr

Claudia Betous-Almeida
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4
France
almeida@laas.fr

Friedemann Bitsch
Institute of Industrial Automation and Software Engineering
University of Stuttgart
Pfaffenwaldring 47
70550 Stuttgart
Germany
bitsch@ias.uni-stuttgart.de

Andrea Bondavalli
Univ. of Firenze
Dip. Sistemi e Informatica
V. Lombroso 6/17
I-50134 Firenze
Italy
andrea.bondavalli@cnuce.cnr.it

Thierry Boyer
Technicatome
BP 34000
13791 Aix-en-Provence Cedex 3
France
tboyer@tecatom.fr

Thomas Bürger
Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen
Universität Stuttgart
Seidenstr. 36
70174 Stuttgart
Germany

Roy B. Carter
NNC Ltd.
Booths Hall
Chelford Road
Knutsford, Cheshire WA16 8QZ
UK

Paul Caspi
VERIMAG
2, rue de Vignate
F-38610 GiLres
France
caspi@imag.fr

Amine Chohra
IEI/CNR
Via Moruzzi 1
I-56100 Pisa
Italy
chohra@iei.pi.cnr.it

Tadeusz Cichocki
Adtranz Zwus
Modelarska 12
40-142 Katowice
Poland,
tadeusz.cichocki@pl.transport.bomardier.com
Felicita Di Giandomenico
IEI/CNR
Via Moruzzi 1
I-56100 Pisa
Italy
digiandomenico@iei.pi.cnr.it

Dacfey Dzung
ABB Corporate Research Ltd.
CH-5405 Baden-Dättwil
Switzerland
dacfey.dzung@ch.abb.com

Rainer Faller
exida.com L.L.C
Wildenholzener Strasse 26
81671 München
Germany
Rainer.Faller@exida.com

Hans R. Fankhauser
Bombardier Transportation, Propulsion & Controls Division
SE-72173 VÄSTERÅS
Sweden
hans.r.fankhauser@se.transport.bombardier.com

John Fox
Advanced Computation Laboratory
Imperial Cancer Research Fund
Lincoln’s Inn Fields
London WC2A 3PX
UK
jf@acl.icnet.uk

Julio Gallardo
Safety Systems Research Centre
Department of Computer Science University of Bristol
Merchant Venturers Building
Woodland Road
Bristol BS8 1UB
UK

Piotr Gawkowski
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
Warsaw 00-665
Poland
gawkowski@ii.pw.edu.pl

Manfred Gingerl
ARCS
A-2444 Seibersdorf
Austria
manfred.gingerl@arcs.ac.at

Janusz Górska
Technical University of Gdańsk
Narutowicza 11/12
80-952 Gdańsk
Poland
jango@pg.gda.pl

Bjorn Axel Gran
OECD Halden Reactor Project
P.O.Box 173
N-1751 Halden
Norway
bjorn.axel.gran@hrp.no

Atte Helminen
VTT Automation
P.O.Box 1301
FIN-02044 VTT
Finland
atte.helminen@vtt.fi

Georg Hoever
Siemens AG
Corporate Technology, CT PP 2
Simulation and Risk Management
81730 München
Germany
Georg.Hoever@mchp.siemens.de
Gordon Hughes
Safety Systems Research Centre
Department of Computer Science University of Bristol
Merchant Venturers Building
Woodland Road
Bristol BS8 1UB
UK
Andrew D. John
NCC Ltd.
Booths Hall
Chelford Road
Knutsford, Cheshire WA16 8QZ
UK
Andrew.John@nnc.co.uk

Ole-Arnt Johnsen
MoreCom
Norway
oaj@morecom.no

Mohamed Kaâniche
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4
France
kaaniche@laas.fr

Karama Kanoun
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4
France
kanoun@laas.fr

Silke Kuball
Safety Systems Research Centre
Department of Computer Science University of Bristol
Merchant Venturers Building
Woodland Road, Bristol BS8 1UB
UK
Silke.Kuball@bristol.ac.uk

Ulrich Laible
Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen
Universität Stuttgart
Seidenstr. 36
70174 Stuttgart
Germany
ulrich.laible@isw.uni-stuttgart.de

Yannick Le Guédart
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4
France

Oliver Mäckel
Siemens AG
Corporate Technology, CT PP 2
Simulation and Risk Management
81730 München
Germany
Oliver.Maeckel@mchp.siemens.de

István Majzik
Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
Műegyetem rkp. 9
H-1521 Budapest
Hungary
majzik@mit.bme.hu

John H. R. May
Safety Systems Research Centre
Department of Computer Science University of Bristol
Merchant Venturers Building
Woodland Road
Bristol BS8 1UB
UK
Christine Mazuet
Schneider Electric
Usine M3
F-38050 Grenoble Cedex 9
France
christine_mazuet@mail.schneider.fr

Martin Naedele
ABB Corporate Research Ltd.
CH-5405 Baden-Dättwil
Switzerland
martin.naedele@ch.abb.com

Odd Nordland
SINTEF Telecom and Informatics
Systems Engineering and Telematics
NO-7465 Trondheim
Norway
Odd.Nordland@informatics.sintef.no

Zsigmond Pap
Dept. of Measurement and Information
Systems
Budapest University of Technology and
Economics
Műegyetem rkp. 9
H-1521 Budapest
Hungary
papzs@mit.bme.hu

Alberto Pasquini
ENEA
Via Anguillarese 301
00060 Roma
Italy
pasquini@casaccia.enea.it

András Pataricza
Dept. of Measurement and Information
Systems
Budapest University of Technology and
Economics
Műegyetem rkp. 9
H-1521 Budapest
Hungary
pataric@mit.bme.hu

Stefano Porcarelli
Univ. of Pisa
Computer Engineering Dep.
Via Diotisalvi 2
I-56126 Pisa
Italy
stefano.porcarelli@guest.cnuce.cnr.it

Günter Pritschow
Institut für Steuerungstechnik der
Werkzeugmaschinen und
Fertigungseinrichtungen
Universität Stuttgart
Seidenstr. 36
70174 Stuttgart
Germany

Felix Redmill
22 Onslow Gardens
London N10 3JU
UK
Felix.Redmill@ncl.ac.uk

Christian Reumann
ARCS
A-2444 Seibersdorf
Austria
christian.reumann@arcs.ac.at

Natacha Reynaud Paligot
Schneider Electric
Usine M3
F-38050 Grenoble Cedex 9
France
natacha_reynaud-paligot@mail.schneider.fr

Antonio Rizzo
University of Siena
Via dei Termini 6
53100 Siena
Italy
rizzo@unisi.it
XVI  List of Contributors

John Rushby
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
USA
rushby@csl.sri.com

Luca Save
University of Siena
Via dei Termini 6
53100 Siena
Italy
save@media.unisi.it

Gerald Sonneck
ARCS
A-2444 Seibersdorf
Austria
gerald.sonneck@arcs.ac.at

Janusz Sosnowski
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
Warsaw 00-665
Poland
jss@ii.pw.edu.pl

Michael Stanimirov
ABB Corporate Research Ltd.
CH-5405 Baden-Dättwil
Switzerland
michael.stanimirov@ch.abb.com

Ioannis Vakalis
EC-Joint Research Centre
Institute for Systems, Informatics & Safety
TP 210
21020 Ispra (Va)
Italy
ioannis.vakalis@jrc.it

Rune Winther
Faculty of Computer Sciences
Østfold University College
Os Allé 11
N-1757 Halden
Norway
rune.winther@hiof.no

Günther Zoffmann
ARCS
A-2444 Seibersdorf
Austria
guenther.zoffmann@arcs.ac.at
Invited Paper

Designing Safety into Medical Decisions and Clinical Processes ............................... 1
John Fox

Reliability Assessment and Security

Security Assessments of Safety Critical Systems Using HAZOPs......................... 14
Rune Winther, Ole-Arnt Johnsen, and Bjørn Axel Gran

Network Security for Substation Automation Systems ...................................... 25
Martin Naedele, Dacfey Dzung, and Michael Sianimirov

A Bayesian Belief Network for Reliability Assessment .................................. 35
Bjørn Axel Gran and Atte Helminen

Safety Case and Safety Analysis

Checking General Safety Criteria on UML Statecharts .................................. 46
Zsigmond Pap, István Majzik, and András Pataricza

Presenting a Safety Case - A Case Study .................................................. 56
Odd Nordland

Safety Functions versus Control Functions ............................................... 66
Hans R. Fankhauser

Medical Systems

A Fail-Safe Dual Channel Robot Control for Surgery Applications .............. 75
Ulrich Laible, Thomas Bürger, and Günter Pritschow

Invited Paper

Modelling the Human in Human Factors .................................................. 86
John Rushby
# Table of Contents

## Human-Machine Interface

**Analyzing Human-Machine Interactions in Safety-Critical Systems: A Basic Applicable Approach** .......................................................... 92  
*Oliver Mäckel and Georg Hoever*

**Analysis of Incidents Involving Interactive Systems** ........................... 100  
*Alberto Pasquini, Antonio Rizzo, and Luca Save*

## COTS – Components off the Shelf

**Experimental Evaluation of Fault Handling Mechanisms** ...................... 109  
*Piotr Gawkowski and Janusz Sosnowski*

**The COTS Debate in Perspective** ......................................................... 119  
*Felix Redmill*

## Testing

**An Investigation on Mutation Strategies for Fault Injection into RDD-100 Models** ................................................................. 130  
*Mohamed Kaâniche, Yannick Le Guédart, Jean Arlat, and Thierry Boyer*

**A Comparison Study of the Behavior of Equivalent Algorithms in Fault Injection Experiments in Parallel Superscalar Architectures** .......... 145  
*Ioannis Vakalis*

**The Effectiveness of Statistical Testing when Applied to Logic Systems** .......... 156  
*Silke Kuball, Gordon Hughes, John H.R. May, Julio Gallardo, Andrew D. John, and Roy B. Carter*

## Formal Methods

**A Classification Scheme for Software Verification Tools with Regard to RTCA/DO-178B** .............................................................. 166  
*Günther Zoffmann, Manfred Gingerl, Christian Reumann, and Gerald Sonneck*

**Safety Patterns – The Key to Formal Specification of Safety Requirements** .......... 176  
*Friedemann Bitsch*

**Formal Support for Fault Modeling and Analysis** ................................. 190  
*Tadeusz Cichocki and Janusz Görski*
# Invited Paper

Project Experience with IEC 61508 and Its Consequences ........................................ 200
*Rainer Faller*

# Control Systems

About the Design of Distributed Control Systems:
The Quasi-Synchronous Approach ............................................................................. 215
*Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot*

Dependability Evaluation - From Functional to Structural Modeling .................. 227
*Cláudia Betous-Almeida and Karama Kanoun*

Tuning of Database Audits to Improve Scheduled Maintenance in Communication Systems .............................................................. 238
*Stefano Porcarelli, Felicita Di Giandomenico, Amine Chohra, and Andrea Bondavalli*

# Author Index

................................................................................................................................. 249
Designing Safety into Medical Decisions and Clinical Processes

John Fox
Advanced Computation Laboratory
Imperial Cancer Research Fund
Lincoln’s Inn Fields, London WC2A 3PX, UK
jf@acl.icnet.uk

Abstract. After many years of experimental research software systems to support clinical decision-making are now moving into routine clinical practice. Most of the research to date has been on the efficacy of such systems, addressing the question of whether computer systems can significantly improve the quality of doctors’ decision-making and patient management processes. The evidence that they can make a major improvement is now clear and interest is beginning to turn to the question of how we can make such systems safe. We outline some example applications and discuss what we can learn in developing safety cases for such applications from the work and experience of the software safety community. Some distinctive challenges also arise in medicine, and some novel safety management techniques to address them are being developed.

1 Introduction

“Many medical errors, which are blamed for up to 98,000 deaths a year in the US, could be prevented according to a report [by the Institute of Medicine]. It points to a series of causes for the errors, from poor handwriting on prescriptions … to doctors struggling to keep up with latest developments.”

http://news.bbc.co.uk/hi/english/health
November 30th 1999.

Many science-based fields are facing a “knowledge crisis”, in that knowledge is expanding explosively while economic resources and human abilities to apply it remain finite. Perhaps the most prominent example is medicine. We are requiring more and more techniques for treating disease and improving our quality of life, yet new expertise is not always quickly disseminated or effectively used. Even in wealthy societies the quality of care is not consistent, and the unprecedented growth in our understanding of diseases and their management is not matched by equivalent abilities to apply that knowledge in practice.

In medicine, as in other fields like the aerospace and power industries, solutions may be found in advanced information technologies for disseminating knowledge and providing active assistance in problem-solving, decision-making and planning and
helping to ensure that complex procedures are carried out in a reliable, timely and efficient way. A variety of technology for supporting clinical decision-making, care processes, workflow etc. are in development (see www.openclinical.org) and there is growing evidence that these systems can significantly improved outcomes. Although the future for these technologies appears bright it is clear that current thinking is focused largely on “efficacy” – there is as yet no safety culture in our field. In the remainder of this paper we give some examples from our own work of the kind of developments which are taking place, and then turn to how we are approaching the safety issue.

2 Computer Support for Clinical Procedures: Two Examples

Drug prescribing is an important area for the use of decision support systems in medicine. Improvements in doctors prescribing decisions could avoid many errors, many of which result in patient harm, and save a considerable fraction of the drugs bill (see http://www.mercola.com/2000/jul/30/doctors_death.htm).

![Fig. 1. A view of the CAPSULE prescribing system showing patient data (top half of rear panel), computer suggested treatment candidates (bottom left) and the explanatory arguments for and against one medication, NAPROXEN (inset panel).](image)
CAPSULE is a decision support system that was designed to assist with drug-prescribing and is shown in figure 1. The top half of the computer display contains a view of a patient record. This highlights the patient’s medical problem (mild osteoarthritis) and information about associated problems, relevant past history, other current drugs, and so on. At the bottom left of the figure there is a list of four possible prescriptions, using the drugs paracetamol, naproxen, ibuprofen and diclofenac. This is a set of candidate medications that CAPSULE has proposed for the treatment of the patient’s condition.

The candidates are displayed in order of relative preference; naproxen, which is second on the list, has been highlighted because the user requires an explanation of this recommendation. In this case CAPSULE suggests that there are four arguments in favor of using naproxen and one against. The argument has been generated because the computer has recognized under “associated problems” a problem of chronic airways obstruction. This is a contraindication for using naproxen.

CAPSULE uses eight types of knowledge for constructing arguments for and against the candidate medications. These deal with:

− whether a drug is contraindicated by other patient conditions
− whether there any interactions with other drugs the patient is taking
− if the patient has already used the drug, does s/he “like it”
− whether the drug has side effects
− if it is recommended in the British National Formulary
− whether it is local policy to use a drug for a specific condition or not
− its cost (high, medium or low)
− whether it is a generic or proprietary drug.

By weighing up the collection of pros and cons which are applicable to the particular patient and circumstances we can place the candidate drugs in a specific order of preference.

A more complex class of application is the management of medical emergencies. A common emergency is an acute asthma attack, which can happen at any time of the day or night and be life-threatening; deaths have been caused through underestimation of the severity of the attack, delays in starting treatment or unsatisfactory subsequent management. In a severe case the threat may develop rapidly and clinicians who are experienced in the management of the condition may be unavailable.

In this setting computers can be useful because they can give advice on the process of care, as well as in decision making like risk assessment and drug prescribing. Figure 2a shows a computer system for the management of acute asthma in which the clinical process is formalised as a network of tasks carried out over time (see top half of figure, note that time flows from left to right). Here the first task is a decision (represented as a circle) whose goal is to determine whether the patient is suffering from a mild, moderate, severe or life-threatening asthma attack, based on criteria established by the British Thoracic Society (BTS).

---

1 Computer Aided Prescribing Using Logic Engineering. CAPSULE was designed by Robert Walton, a general practitioner, in collaboration with Claude Gierl, a computer scientist.

2 Developed with David Elsdon, Claude Gierl, and Paul Ferguson.
In figure 2b the decision has been “opened” in a panel at the top to reveal its internal structure. Here the decision candidates are severity levels (mild, moderate etc. c.f. medications as in CAPSULE) and a reminder panel show the data that the BTS
says are relevant to formulating the arguments for the different severity levels. Additional panels and widgets call be used for data entry, such as the “peak expiratory flow rate meter” which is shown, and various other clinically useful dialogues.

Once the decision is taken (the patient is suffering a moderate asthma attack shown by a tick on the right of the box) the computer moves on to the next task, represented by the small rounded rectangle marked “mild and moderate management” in the overview task network (figure 2a). This is a plan, containing a number of data acquisition and assessment tasks and treatment decisions, and this procedure is continued, task by task, prompting the clinician to record information, take decisions and actions etc, until the process is complete, a period which will typically be a couple of hours.

Despite correct care a patient may not respond adequately to treatment, or may even deteriorate unexpectedly, so the asthma management system has been equipped with a “hazard detection” mechanism, represented by the two additional decisions (circles) at the centre of figure 2a, near the bottom. These are “monitors” or “watchdogs” which operate autonomously, checking clinical data without user involvement. Their purpose is to monitor the patient state for potentially hazardous events or trends. If a serious hazard arises the watchdogs will raise an alarm or take some other appropriate action.

A range of medical applications are described in detail in Fox and Das (2000), together with details of PROforma, a technology for implementing decision support and workflow systems using logic programming and other AI technologies.

3 Ensuring Safety in Clinical Processes

There is now considerable evidence that computer systems can have a very real benefit in improving patient care, so much so that many groups are developing technologies for such purposes (see www.openclinical.org for outline descriptions of the main developments and links to the relevant research groups). Exciting as these new developments are there is a downside that a software safety audience will immediately recognise. It is one thing to develop a medical intervention (such as a drug which is efficacious against an abnormal condition or a virus or a tumour) it is another to be sure that the intervention has no dangerous side effects or other adverse reactions during operational use. In our book we also discuss a range of safety issues that can arise in clinical settings and how we are trying to address them using a range of established and some novel techniques. The rest of the paper provides a very brief outline of this material.

All medical technologies, including information technologies, involve potential risks. Wyatt and Spiegelhalter (1991) argue along traditional lines that decision support systems and other technologies should undergo rigorous trials before they are made available for general use. We are inclined to go further and argue that we should ensure that such technologies are designed to be safe, with harmful side effects or adverse consequences of use kept to an absolute minimum. In this respect the medical informatics community has much to learn from the software engineering and critical systems engineering communities.
3.1 Lessons from Software Engineering

The first important lesson concerns development lifecycles. Nowadays large software systems are commonly developed within structured lifecycles to achieve quality through systematic design, development, testing and maintenance processes. Structured lifecycle models have been developed for AI systems, as in KADS, a methodology for knowledge based systems that provides tools and techniques to support design, specification, knowledge acquisition, knowledge reusability, verification etc. (Wielinga, Schreiber and Breuker, 1992). Figure 3 illustrates the development lifecycle that we have adopted. This covers the basic design, development and operation that we have found to be well suited to the development of decision and workflow support systems in medicine.

There has also been growing interest in applying techniques from formal software engineering to AI in recent years. Some researchers have explored the use of mathematical techniques for specifying and proving the soundness of knowledge based systems, inspired by formal specification languages e.g. (ML)² (van Harmelen and Balder, 1992). The motivation for adopting formal design techniques has been the desire to remove many of the apparently ad hoc practices associated with AI systems development, and to provide techniques for automated verification and validation of
the system knowledge base. We have adopted both these methods in our technology for developing our applications systems.

PROforma is a specification language for describing processes, such as clinical processes. It provides a set of standard constructs, “tasks”, that are appropriate for describing processes in terms of the plans, data, actions and decisions that are required in order to achieve a medical goal (e.g. the therapeutic goals inherent in a care plan). The method is based on a set of task objects as illustrated in figure 4.

Tasks are formal software objects that can be composed into networks representing plans or procedures that are carried out over time, and which incorporate decision making and contingency management rules to deal with situations and events if and when they occur. The asthma care pathway in figure 2a and figure 2b is a moderately complex example of such a process. The PROforma task set is supported by a set of reusable software components which can be assembled or composed into a task network, using specialized CASE tools. The CASE tools generate a specification of the application process in the PROforma representation language.

Since we have a formal model of the general properties of decisions, plans and other PROforma tasks there is considerable scope for syntax-directed and other kinds of model checking to ensure the integrity of a specification. Once all the recognizable

---

**Fig. 4.** The PROforma task ontology. Agent expertise models are composed out of these networks, forming networks of tasks carried out reactively in response to events (actions, decisions etc) and deliberatively to achieve goals (plans). The PROforma method and toolset is described in Fox et al, (1997) and see example A at end of article. The underlying formal agent model is described in Das et al (1997).
syntactical and other logical errors have been removed it is possible to test the specification in order to its operational interpretation. Tasks are enacted according to a well-defined control regime in which tasks pass through a sequence of states, the particular sequence being determined by the situations and events that are encountered during operation.

The PROforma development environment has proved to be quite successful in permitting complex clinical and other processes to be modelled, checked and tested rapidly and confirming that the behaviour is that which is intended. Correct specification is not, of course, safe specification so we have also sought to learn lessons from the safety engineering community in developing the PROforma method.

Traditional safety engineering involves two main kinds of activity: (1) analyzing faults and the hazards they give rise to, and (2) incorporating techniques to reduce the likelihood of faults and to prevent hazards turning into disasters. Leveson (1995) translates these activities into a special lifecycle for safety critical applications. Safety

Fig. 5. Agent development lifecycle augmented with a parallel safety lifecycle involving hazard analysis and removal.
needs to be considered throughout the lifecycle; and it should be considered separately from issues of cost, efficiency and so forth. The same lessons apply to AI methods like PROforma. We are therefore exploring how to introduce a separate safety design process into our life cycle, as shown in figure 5.

Engineering a software system of any kind to be safe broadly consists of ensuring that the specification and implementation are sound (stream 1 on the left) and trying to anticipate all the hazards that might arise during, whether due to internal system faults or the environmental threats that can occur, and building appropriate responses into the hardware and software to ensure continued functioning, or failsafe shutdown (safety stream on the right). A variety of methods for analysing systems and their expected operation are available, and an important part of our effort concerns building standard techniques such as HAZOP (Redmill et al, 1999).

With the wide range of techniques now available, much can be done to ensure that an operational system such as a PROforma application will behave effectively as intended. However we can rarely, if ever, guarantee it. Even with a rigorous design lifecycle that incorporates explicit hazard analysis and fault eradication techniques there is a residual weakness for both AI and conventional software. The strategy depends upon the design team being able to make predictions about all the circumstances that may hold when the system is in routine operation.

In many fields it is possible to anticipate most of the hazards that can arise, but in medicine and other complex settings this seems to be out of the question. The scope for unforeseen and unforeseeable interactions is vast. The environments in which the software is used may be quite different from those envisioned by the designers. There may be unexpected side effects if actions are correctly carried out but in unanticipated conditions, or two or more actions taken for independently justifiable reasons may have dangerous interactions. It is simply not possible to guarantee that all possible hazards will be exhaustively identified for substantial applications.

3.2 Can Safety Engineering Learn Anything from Medicine?

“The old admonition about ‘the best-laid plans of mice and men’ also applies to the best-laid plans of computers”

David E. Wilkins, 1997, p 305.

Rather than try to anticipate all the specific hazards that can arise in a clinical setting, which is doomed to fail, an alternative strategy may be to provide the software with the operational ability to predict hazardous consequences prior to committing to actions, and to veto actions or preempt hazards when a potentially dangerous trend is recognized. The idea that we have been interested in for a long time is that of applying AI methods for reasoning, problem solving and similar methods to managing situations as they occur and finding remedies that are appropriate to the context (Fox, 1993).

An early example of this approach was the "safety bag expert system" (Klein 1991) which was designed to manage the routing of rolling stock through the shunt yards at Vienna station. The system’s goal was to plan safe routings for rolling stock being moved through a busy rail network.
Planning the shortest or other minimum-cost route through a rail network is clearly hazardous. A section of the route may have other wagons on it, or points (switches) might be set such that another train could enter the section. Klein’s safety bag expert system had a dual channel design in which one program proposed viable routes through the tracks and points while a second system monitored the proposed routes and assessed them for potential hazards.

The safety bag is a rule-based system in which the rules’ conditions embody knowledge of the safety regulations that apply at Vienna station. The actions of the rules are to veto (or commit to) routes proposed by the route planner. The use of rules to express what is, and is not, acceptable behavior brings together the documentation and implementation of a safety-critical system. The “safety policy” embodied in the rules is explicit and readable for both the original designers and independent inspectors. The rule set is also executable, and the software will operate according to these rules so we can be more confident that the safety policy will be followed than if the rules were no more than the designers’ documented intentions.

The safety bag is a novel approach to ensuring that software can be made safe. However, the concept has three significant limitations.

First, the rules simply say “if such and such is true then do this, but if such and such then don’t do it”. The rationale behind the rules and the safety goal(s) that are implied are not explicitly represented. If a rule turns out to be inappropriate for some reason, the system has no way of knowing it.

Second, the rules of the protocol are “special case” regulations that are specific to the domains of trains: they do not capture general principles. It would be desirable to have a generalized collection of safety rules that could be applied in a range of applications.

Finally, we would like a general theory of safety that would provide the foundations for specifying general safety protocols that might be used in any domain, from medicine to train routing, to autopilots to traffic management systems (Fox, 1993). There would be significant benefits if we could formalize general safety knowledge separately from a software agent’s domain-specific knowledge.

Medicine is a field in which there is daily experience of safety management, indeed clinicians are arguably among the most skilled and knowledgeable people when it comes to developing strategies for managing hazards. Although clinicians do not typically discuss hazard management in generalised terms there do appear to be general principles that are in practice applied routinely.

Some years ago my colleague Peter Hammond reviewed a large number of cancer treatment protocols with the aim of establishing general principles of good care and capturing them in a logic program. Not only did he succeed in doing this (Hammond et al, 1994) the principles that he identified seem to be potentially applicable in a wide range of domains. Hammond carried out a detailed review of more than 50 cancer treatment protocols (formal documents setting out “best practice” in the management of different cancers) in order to identify material that dealt with some aspect of patient safety. This material was extracted as text statements that were analysed to establish implicit safety rules. These rules were then formalised in terms of generalised if…then… rules about the care of patients. Although these rules were identified from a medical study they do not specifically refer to specific features of the cancer domain.

---

3 In the light of the recent tragedy at Paddington Station in London this example seems particularly timely, though that was an error in plan execution rather than in route planning.
or even medicine. This suggests that the rules may capture general principles that may be used in other application domains where we need to anticipate, avoid, detect and manage hazards. Here are two examples.

**Anticipate adverse events with prophylactic actions**
It is often possible to anticipate hazards by taking suitable safety precautions that help to prevent them or at least to diminish their undesirable side effects. Anticipation of likely hazards is common in many cancer protocols.

- Prehydration helps avoid dehydration due to vomiting induced by chemotherapy.
- Folinic acid rescue helps ameliorate methotrexate-induced bone marrow suppression.
- Prophylactic antibiotics help avoid infection due to bone marrow suppression.

A logical representation of the underlying principle is as follows:

\[
\text{If} \quad \text{Action1 is necessary part of Plan and Action1 produces Effect and Effect is potentially hazardous and Action2 helps avoid Effect and Action2 is compatible with Plan} \\
\text{Then} \quad \text{Action2 should be performed to anticipate Effect of Action1 in Plan}
\]

**Avoid augmenting hazardous side-effects**
It is important to identify actions that might exacerbate predictable hazards – for example, the potential damage to kidney function from chemotherapy.

*Nephrotoxic antibiotics such as gentamicin should be avoided during and immediately after giving cisplatin.*

*Cytarabine is incompatible with fluorouracil*

Generalizing, we have:

\[
\text{If} \quad \text{Action1 is a necessary part of Plan and Action1 produces Effect and Effect is potentially hazardous and Action2 exacerbates or makes Effect more likely and Action2 has alternative without Effect} \\
\text{Then} \quad \text{Action2 should not be performed during Action1 in Plan}
\]

The main safety principles that were identified in Hammond’s review are summarized informally below.
ANTICIPATE: Prevent or ameliorate known hazards before executing actions.

ALERT: Warn about hazards arising from inadequate execution of actions.

AVOID: Avoid (extraneous) actions likely to exacerbate hazards due to actions.

AVOID: DIMINUTION: Avoid (extraneous) actions likely to undermine the benefits of essential actions.

MONITOR: Monitor responses which herald adverse events or hazardous situations.

SCHEDULE: Schedule actions in time for best effect and least harm.

REACT: React appropriately to any detected hazard.

These clearly represent valid, even common sense, rules of safe operation. If they form a routine part of clinical practical then surely they can also be embodied in software systems. In our book we discuss ways in which these principles can be embodied in PROforma types of clinical applications, and intelligent software agents in general.

4 Conclusions

In recent years knowledge engineers have become much more concerned with quality of design and implementation than traditionally, and they have learned much from conventional software engineering in this process. In return AI may have some novel techniques to offer which could add a further level of safety to safety-critical systems by adding “intelligence” into the designs. Alongside the pursuit of formal lifecycles, rigorous specification of software etc. have investigated the idea of active safety management techniques which deal with hazards that arise unexpectedly during system operation.

References

Fox, J.: On the soundness and safety of expert systems. Artificial Intelligence in Medicine, 5 (1993) 159-179


Security Assessments of Safety Critical Systems Using HAZOPs

Rune Winther¹, Ole-Arnt Johnsen², and Bjørn Axel Gran³

¹ Faculty of Computer Sciences, Østfold University College,
Os Allé 11, N-1757 Halden, Norway.
rune.winther@hiof.no
² moreCom, Norway
oaj@morecom.no
³ Institute for Energy Technology, Norway.
bjorn.axel.gran@hrp.no

Abstract. Concerned with serious problems regarding security as a safety issue, a HAZOP specifically suited for identifying security threats has been developed. Unfortunately, the emphasis placed on security issues when developing safety critical systems is to often inadequate, possibly due to the lack of “safety-compliant” security methods. Having had the opportunity to adapt the HAZOP-principle to the security context, a HAZOP was established which is well-suited for handling security issues in a safety context. Indeed, since the main modification of the method consists of establishing new guidewords and attributes, it is quite possible to handle security issues as part of the traditional hazard analysis. In addition, while presenting the modified HAZOP-method, its use on safety related systems will be demonstrated.

1 Introduction

Increasing dependence on programmable equipment (or Information and Communication Technology, ICT) is a well known fact. Systems used in, for example, transportation and process control systems involve exposure to the risk of physical injury and environmental damage. These are typically referred to as safety-related risks. The increased use of ICT-systems, in particular combined with the tendency to put “everything” on “the net”, gives rise to serious concerns regarding security, not just in relation to confidentiality, integrity and availability (CIA), but also as a possible cause of safety problems. With the increasing dependence on ICT-systems saboteurs are likely to use logical bombs, viruses and remote manipulation of systems to cause harm. Some simple examples illustrate the seriousness:

- The result of a HIV-test is erroneously changed from positive to negative due to a fault in the medical laboratory’s database system.

¹ Security is in this context interpreted as the systems ability to uphold confidentiality of information, integrity of information/systems and availability of information/services [5].
– The next update of autopilot software is manipulated at the manufacturers site.
– The corrections transmitted to passenger airplanes using differential GPS (DGPS) as part of their navigation system is manipulated in such a way that the airplane is sent off course.

Note that security might be a safety problem whether the system is real-time or not, and that it is not only the operational systems that need protection. Systems under development and software back-ups could also be targeted by an attacker.

It is our impression that in the past, and to a great extent at present, most safety-related assessments don’t seem to include proper considerations of security as a safety problem. One possible reason might be the lack of methods which can be used to identify security threats in a safety context. Although there do exist analytical techniques from both the security and the safety traditions, the approaches used within these areas seem to be different. A “convergence” of methods would therefore be beneficial.

Based on experience from using safety-related techniques in security projects, we have the opportunity to “think safety” in a security context. Related to the development of a risk analysis handbook (security issues) for Telenor, two of the authors were involved in an evaluation of methods such as Fault Tree Analysis (FTA), Event Tree Analysis (ETA), Failure Mode Effect Analysis (FMEA) and HAZard and OPerability studies (HAZOP) for use in security. One of the conclusions was that the HAZOP principle seemed well suited, assuming that adequate guidewords could be established.

We will in this paper present a “security-HAZOP” which has emerged from our experiences in practical projects and demonstrate its use in a safety context. Finally, we will point to a new EU-project which objective is to combine e.g. HAZOP with object oriented modeling and the use of UML in the development of security-critical systems.

2 What Is HAZOP?

A HAZOP study is a systematic analysis of how deviations from the design specifications in a system can arise, and whether these deviations can result in hazards. The analysis is performed using a set of guidewords and attributes. The guidewords identified in for use when analysing programmable electronic systems (PES) are no, more, less, as well as, part of, reverse, other than, early, late, before and after. Combining these guidewords with attributes, such as value and flow, generic deviations can be described thus providing help in identifying specific safety related deviations.

A HAZOP study is typically conducted by a team consisting of four to eight persons with a detailed knowledge of the system to be analysed. The HAZOP-leader of the group will normally be an engineer with extensive training in the

\(^2\) Telenor is Norway’s largest telecommunication company.
\(^3\) Winther and Johnsen
use of HAZOP and other hazard analysis methods. The analysis itself is done by going systematically through all system components identifying possible deviations from intended behaviour and investigating the possible effects of these deviations. For each deviation the team sets out to answer a series of questions to decide whether the deviation could occur, and if so, whether it could result in a hazard. Where potential hazards are detected, further questions are asked to decide when it might occur and what can be done to reduce the risk associated with the hazard.

3 Adapting the HAZOP to a Security Context

In this chapter we will briefly discuss why we have chosen to use HAZOPs for identifying security threats and then present the proposed modifications to the original HAZOP.

3.1 Why HAZOP?

Even though HAZOPs originally were developed for use in a specific context, namely the chemical industry [1], experience over the years has shown that the basic principle is applicable in different contexts. [2,6] presents modified HAZOPs for use on systems containing programmable electronics. The fact that HAZOPs see widespread practical use in diverse areas indicates that it is a good candidate for identifying security threats. After all, the aim is the same in security as in safety: We want to identify critical deviations from intended behaviour. There are also other arguments that lead to the same conclusion. Comparing HAZOPs to FMEA (which is a possible alternative) we see that FMEAs are best used in situations where the analysis can be performed by one or two persons and where the identification of possible failure modes is not too complicated. As the FMEA (at least in principle) requires that all failure modes of all components must be scrutinized equally thoroughly, we expect problems in using the method when dealing with complex computerized systems, having a multitude of possible failures, and usually requiring that more than two persons participate if all relevant aspects shall be adequately covered. This does not imply that we discard FMEA as a possible method for analysis of security threats. In situations where the possible failure modes are relatively obvious and the aim of the analysis is more focused on consequences, FMEA is probably a good candidate. Having a well structured description of the system might be one way of achieving this.

3.2 Modifying the HAZOP to Identify Security Threats

Since the HAZOP principle obviously should remain the same, our focus has been on identifying guidewords and attributes which will help us identify security-related deviations. As we are primarily focusing on the CIA of security, i.e. confidentiality, integrity and availability, the intuitive approach is to define these
as attributes and then continue by evaluating whether the guidewords defined in [6] (see Chapter 2) can be used, or if new ones are needed.

When systematically combining the guidewords in [6] with each of the CIA attributes, it is quickly realized that many combinations doesn’t seem to be useful. For instance, although “more confidentiality” could be interpreted as too much confidentiality, implying that information is less available than intended, this deviation is more naturally identified through the statement “less availability”. Since our prime concern is that the level of confidentiality, integrity and availability won’t be adequate, a pragmatic evaluation suggests that only “less” is useful. However, only considering the applicability of preexisting guidewords is not enough. We need to see if there are other suitable guidewords. An interesting question is: “What causes inadequate CIA?” Firstly, the loss of CIA might happen both due to technical failures and human actions. Furthermore, typical security threats include deliberate hostile actions (by insiders or outsiders) as well as “trivial” human failures. It makes sense, therefore, to include guidewords encompassing the characteristics deliberate, unintentional, technical, insider and outsider. In order to be able to combine these with the CIA attributes in a sensible way, we have chosen to use negations of the CIA attributes, i.e. disclosure, manipulation and denial. Furthermore, since e.g. deliberate actions might be by both insiders and outsiders, we see that it might be beneficial to combine more than one guideword with each attribute. To accommodate this we have chosen to structure the HAZOP expressions as illustrated (with examples) in Table 1.

Table 1 summarizes the guidewords and attributes we suggest as a starting point when using HAZOPs to identify security threats.

Table 1. A new way of combining guidewords and attributes, together with some simple examples.

<table>
<thead>
<tr>
<th>Pre-Guideword</th>
<th>Attribute</th>
<th>of</th>
<th>comp.</th>
<th>due to</th>
<th>Post-Guideword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliberate</td>
<td>manipulation</td>
<td>of</td>
<td>firewall</td>
<td>due to</td>
<td>insider</td>
</tr>
<tr>
<td>Unintentional</td>
<td>denial</td>
<td>of</td>
<td>service</td>
<td>due to</td>
<td>technical failure</td>
</tr>
</tbody>
</table>

Table 2. Basic guidewords and attributes suitable for identifying security threats

<table>
<thead>
<tr>
<th>Pre-Guideword</th>
<th>Attribute</th>
<th>Post-Guideword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliberate</td>
<td>Disclosure</td>
<td>Insider</td>
</tr>
<tr>
<td></td>
<td>Manipulation</td>
<td>Outsider</td>
</tr>
<tr>
<td>Unintentional</td>
<td>Denial</td>
<td>Technical failure</td>
</tr>
</tbody>
</table>
It is important to note that in each analysis it will be necessary to evaluate what guidewords and attributes that are suited. Both the attributes and the post-guidewords given in Table 2 are rather generic and could be refined to better describe relevant deviations. For instance, the attribute manipulation could be replaced by removal, alteration, fabrication, etc. While the post-guidewords listed above define some generic threat agents, these could be replaced or augmented by describing the possible techniques these threat agents might use. Spamming, social manipulation and virus are relevant examples. Using these more specific attributes and guidewords, we obtain the examples in Table 3. In the first example the guidewords unintentional and virus are combined with the attribute fabrication and applied to the component mail.

Table 3. Examples of more specific expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Possible security threats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unintentional fabrication of mail due to virus</td>
<td>Improper handling of mail attachments. Inadequate virus protection.</td>
</tr>
<tr>
<td>Deliberate disclosure of patient records due to social manipulation</td>
<td>Improper handling of requests for information from unknown persons.</td>
</tr>
</tbody>
</table>

If we replace unintentional with deliberate in the first example, achieving the expression Deliberate fabrication of mail due to virus, we immediately associate this with an attacker using viruses of the “I LOVE YOU” type to cause harm. Although the threats we have identified for the component mail are closely related, changing from unintentional to deliberate moves our focus from sloppy internal routines to hostile actions. Table 4 provides an extended list of guidewords and attributes compiled through various projects.

Table 4. An extended list of guidewords and attributes suitable for identifying security threats.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Post-Guidewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>disclosure, manipulation, disconnection, fabrication, delay, corruption, deletion, removal, stopping, destabilisation, capacity reduction, destruction, denial</td>
<td>insider, outsider, technical failure, virus, ignorance, fire, faulty auxiliary equipment, sabotage, broken cable, logical problems, logical attack, planned work, configuration fault, spamming, social manipulation</td>
</tr>
</tbody>
</table>
Going through the list of attributes and guidewords one will see that some of them have similar meanings. However, although some of the words can be considered quite similar, they might give different associations for different people. Furthermore, words that have similar meanings in one context might have different meanings in another. Taking “deletion” and “removal” as an example, it is easily realized that in an analysis of physical entities the word “removal” might give sensible associations, while “deletion” doesn’t.

Having discussed possible guidewords and attributes we also need to discuss what a typical component will be in the context of the security-HAZOP. In the original HAZOP the components typically are physical entities such as valves, pipes and vessels. In the “PES-HAZOP” described in [6], the entities might be both physical and logical. Since the main focus of the security-HAZOP is on possible threats to confidentiality, integrity and availability, components must constitute entities for which these attributes are meaningful. While confidentiality is relevant for information, integrity and availability are relevant in respect to both information and functionality. Hence, we suggest that the focus of the security-HAZOP should be on the various types of information handled by the system, and on the functions the system perform. In fact, it could be argued that we could limit the selection of components to the information components, since any failure of an ICT system must in some way be related to changed, erroneous or missing information. In practice, however, we will include both information, functions and in some cases even physical entities in our list of components as they provide different perspectives. In some situations it might be more intuitive for the experts to consider physical entities than the more abstract information types. Deciding which components to analyse should be done pragmatically, where available time and experience are relevant factors to consider.

An important difference between physical entities and information is that the latter are not bounded to be at a single place at any one time (although they can be). Information might be stored in several locations as well as being in transition between various nodes in a network. Although functionality is naturally associated with physical entities they are not necessarily limited to a single entity either. The function “file transfer”, for instance, is a functionality that involves at least two physical entities. The reason for pointing out these more or less obvious facts, is that they have affected the analyses we have performed. Let’s illustrate this with a simple example: Consider a system consisting of a client and a server where the client requests a download of a patient record. Relevant components in this scenario are the patient record (information) and information transfer (function). Physical entities are the two computers, which the client and server software are running on, and the network which connects the two. If we were to specifically cover all physical entities, as well as information types, we would have to evaluate possible threats to the patient record at the client computer, the server computer and in the network. Since threats exists for all of these this is not irrelevant. However, it might become tedious in cases where there are many physical entities. An alternative approach consists of

---

4 Which can be subdivided into a number of components.
evaluating threats to the information detached from the physical entities. Since threats have to be related to the information we don’t necessarily miss relevant threats, although more emphasis will have to be put on the cause-consequence analysis. It should be noted that unavailability of information, which could be caused by threats to the physical entities, must be included in the list of threats to the information. In practice, a pragmatic approach will be to use a mixture, thus putting specific focus on selected parts of the system by considering some physical entities in more detail than others.

4 Practical Use of the “Security-HAZOP”

In this chapter we will demonstrate the method’s applicability on a safety related system.

The basic elements of the system referred to as a “Train Leader (TL) Telephone System” (TLT-system) are shown in Figure 1. The TL’s main task is to supervise the traffic and ensure that signals etc. are correct. The safety aspects of this system have been analysed using both HAZOP and FMEA and the experiences are in the proceedings of an earlier SAFECOMP [4]. The analysis of security illustrated below was never done in practice, although the results might indicate that it should have been.

The TLT-system’s main functions are:

– To present incoming calls from trains at the train leader’s (TL’s) terminal (PC), including information regarding the trains’ positions.
– Connect TL’s telephone to whatever incoming call the TL selects.
– Set up outgoing calls as requested by TL.
– Route incoming calls from trains to the TLs responsible for the various trains.

The TL’s main task is to supervise the train traffic and ensure that signals etc. are correct. Since the TLs are authorized to give “green-light” to trains in the case of signal system failure it is important that calls are connected to the correct TL and that the information presented to the train leader is also correct. Erroneous routing of calls, or misleading information regarding the trains’ identity or position, could cause serious accidents.

In this illustration, we will focus on one specific scenario, namely: “Train driver initiates a train radio call to TL and TL answers call”. The analysis will be performed by going through the following steps:

1. Identify relevant components.
2. Construct relevant expressions based on the suggested guidewords, attributes and components.
3. Evaluate whether the expressions identify relevant security threats.

In the scenario we are investigating we have the following sequence of messages:
1. Train identifier (ID) and train position are sent from TRB to TRX. Train ID is obtained from the on-board radio while train position is determined from sensors placed along the tracks.
2. Train ID and train position are sent from TRX to TLT-server.
3. Train ID and train position are sent from TLT-server to the appropriate TL PC.
4. When TL decides to answer the call the TL PC sends a connect command to the TLT-server.
5. TLT-server commands TRX and PBX to connect the incoming call to TL’s telephone.
6. TLT-server updates TL’s PC-screen.

As noted in Section 3.2, typical components for the security-HAZOP are information types and functions. From the scenario above we see that we have three types of information: Train ID, train position and voice. The most critical functions are routing of voice and call information to correct TL, and to present call information at the TL’s terminal. Train ID originates from the train itself and is sent through TRB, TRX and TLT-server before it is presented to the TL. The train ID is used by the TLT-server to decide which TL should receive the call.

For simplicity we have chosen to ignore the post-guidwords and to focus on deliberate actions related to manipulation and denial of train ID and voice. Table 5 presents both the constructed expressions, security related hazards and possible causes. It should be noted that this table is not a complete list of relevant hazards. Erroneous train position is obviously another critical failure that could potentially be caused by an attacker.

Having identified security threats in the TLT-system, the next activity is to evaluate whether these can cause hazards to occur. From this example we
Table 5. Examples of the use of guidewords together with typical results

<table>
<thead>
<tr>
<th>Expression</th>
<th>Threat</th>
<th>Causes</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliberate manipulation of train ID.</td>
<td>Train ID is altered.</td>
<td>TRB, TRX or communication links between TRB/TRX or TRX/TLT-server has been manipulated.</td>
<td>Call information is wrong. Call is routed to wrong TL.</td>
</tr>
<tr>
<td>Deliberate denial of train ID</td>
<td>Communication between train driver and train leader is inhibited.</td>
<td>TRB, TRX or cabling has been manipulated, destroyed or in any other way forced to fail.</td>
<td>Train cannot be given a manual “green-light”. Emergency calls cannot be made.</td>
</tr>
<tr>
<td>Deliberate manipulation of voice.</td>
<td>Unauthorized person responds to call from train and impersonates a TL.</td>
<td>TRB, TRX, PBX or com. links in between has been manipulated to connect unauthorized person to a call from a train.</td>
<td>Manipulation of train driver to perform unsafe action.</td>
</tr>
</tbody>
</table>

see that the threat “Train ID is altered”, which has the possible consequences “Call information is wrong” and “Call is routed to wrong TL”, is naturally associated with the hazard “Wrong train receives green-light”. In the analysis actually carried out for the TLT-system the possible causes for this hazard were limited to internal software and hardware failures, thus illustrating the limited scope of the analysis.

Let us now make a simple comparison with some combinations of guidewords/attributes for the TLT-system based on the PES-HAZOP [6]:

- Train ID combined with Other Than
- Train ID combined with No

Clearly, applying these guidewords does not mean that we will not identify security threats. Manipulation of Train ID is one possible cause of getting an erroneous Train Id. The benefit of applying the security specific guidewords and attributes is that our attention is specifically directed to the security issues, thus reducing the possibility of missing out on important security threats. While the guidewords in the PES-HAZOP tends to focus on system failures, the security-HAZOP emphasizes the systems vulnerability to human actions and incorrect information.
5 A Framework for Efficient Risk Analysis of Security-Critical Systems

The successful employment of HAZOP and FMEA to identify and analyse safety risks in the TLT system [4] is one of the arguments for the IST-project CORAS [3]. The CORAS main objective is to develop a practical framework, exploiting methods for risk analysis developed within the safety domain (such as HAZOP), semiformal description methods (in particular, methods for object-oriented modelling), and computerised tools (for the above mentioned methods), for a precise, unambiguous and efficient risk analysis of security-critical systems. One hypothesis considered in the project is that a more formal system description can make it easier to detect possible inconsistencies. Another main objective is to assess the framework by applying it in the security critical application domains telemedicine and e-commerce. We believe that the security-HAZOP presented in this paper, sketching out how new guidewords, attributes and a new template could be made, can be another input to this project. The fact that the project has got funding from January 2001, and will run for 30 months, is also an example of a growing awareness with respect to the identification of security threats in safety critical systems.

6 Conclusions

We have shown that it is possible to adapt the HAZOP-principle for analysis of security. The adaptation required new guidewords, new attributes and a new template for combining guidewords and attributes. Since the HAZOP-principle is well known in the “safety world”, extending the HAZOPs already in use with the modifications presented in this paper should enable a relatively easy incorporation of security analyses in safety contexts.

We have argued that relevant components to be analysed could be limited to the information types handled by the system. Since the same information might be stored in several locations, as well as being in a state of transfer, systematically going through all physical entities for each type of information quickly becomes tedious, without necessarily improving the threat identification.

References

Network Security
for Substation Automation Systems

Martin Naedele, Dacfey Dzung, and Michael Stanimirov

ABB Corporate Research Ltd.,
CH-5405 Baden-Dättwil, Switzerland,
{martin.naedele,dacfey.dzung,michael.stanimirov}@ch.abb.com
Tel +41 56 486 8339
Fax +41 56 486 7365

Abstract. The protection of critical infrastructure against electronic and communication network based attacks becomes more and more important. This work investigates the threat of network-based attacks on substations, the nodes of the electric power grid. Three fundamental types of attacks are derived and a secure communication protocol is proposed to counter these attacks by reducing them to a failure mode that can be dealt with similar to other, non-malicious subsystem failures by safety mechanisms.

1 Introduction

1.1 Motivation

The protection of critical infrastructure against electronic and communication network based attacks becomes more and more important for utilities nowadays [12, 5].

- In many parts of the world the market situation has changed and led to higher competition between utilities and at the same time smaller utility company sizes and therefore decreasing ability to support the expenses and expertise necessary for issues such as security. This increases the risks of security incidents.

  *Competition, a first for power suppliers, has created what IEEE-USA calls “financial incentives for malicious intrusion into computers and communications systems of the electric power industry and marketplace participants.” [5]*

- Electronic attacks, also called “information warfare”, have become an established means of destabilization in modern day conflicts. Power substations are among the most vulnerable points in the electricity infrastructure [9] and therefore a prime target for these kinds of attack, both by hostile governments and terrorist organizations [15].
Water, electricity, […] and other critical functions are directed by computer control […]. The threat is that in a future crisis a criminal cartel, terrorist group, or hostile nation will seek to inflict economic damage […] by attacking those critical networks. […] The threat is very real. [2]

Proprietary field buses in substation automation systems will more and more be replaced by open, standard protocols such as Ethernet which raises additional concerns.

1.2 Previous Work

While there exists a huge amount of research on security in home and office information systems - [14] gives a good introduction into the topic, only very little has been done in the area of network security for automation systems. [11] investigates the suitability of the firewall concept for remotely accessing information systems and proposes a smartcard based infrastructure for encryption and digital signatures. In [10] security issues with regard to remote access to substation automation systems are analyzed and some security measures are proposed, with a particular emphasis on the use of passwords and proper selection of password. Both papers are concerned with remote access via public networks, not with malicious devices inside the automation system.

1.3 Contributions

This paper reports the results of a security analysis of an Ethernet-based substation automation communication network. The main contributions presented are the following:

– It is shown that by threat scenario analysis all types of possible attacks can be classified into three categories: message insertion, modification, and suppression.
– A communications protocol is proposed which reduces these three types to one, message suppression. Message suppression has consequences that are very similar to component failures which can be dealt with using standard fault-tolerance strategies.

1.4 Substation Automation Systems

Power generation, transmission, and distribution are a fundamental need of our society. Power grids of different topologies are responsible for transporting energy over short or long distances and finally distributing it to end-consumers such as households and companies. The nodes in a power grid are called substations and take over the voltage transformation and/or the routing of energy flow by means of high-voltage switches (circuit breakers, disconnectors, earthers). Substations consist of several bays which house the inputs and outputs towards the grid and
one or more busbars which connect these bays. Substations may be manned or unmanned depending on the importance of the station and also on its degree of automation. Substations are controlled by Substation Automation Systems (SAS). Since unplanned network outages can be disastrous an SAS is composed of all the electronic equipment that is needed to continuously control, monitor, and protect the grid. The protection functionality is especially critical, both for the substation itself and for the grid. Therefore, a SAS uses different, redundant protection schemes. An SAS can be classified as a distributed soft real-time system with required response times between 10ms and 100ms. It is usually composed of 20...100 Intelligent Electronic Devices (IED) which are connected by a communications network. Any device that can run executable program code and provides a communications interface would be classified as an IED. While some real-time critical functions are executed more or less autonomously on a single IED, other functions are realized in distributed form over many IEDs.

1.5 Overview

The paper is structured as follows: In the next section an example architecture for a substation automation communication network will be introduced on which the threat analysis of section 3 is based. In section 4 various countermeasures to these threats will be discussed. Section 5 summarizes our findings and concludes the paper.

2 Substation Automation System Architecture

A realistic example network topology for one bay of a substation is shown in Fig. 1.

The network uses optical, switched Ethernet. With switched Ethernet exactly one network device is connected to each port of the switch and therefore to each Ethernet segment. In comparison to a bus topology this has several advantages:

- It avoids collisions on the bus and therefore improves determinism while at the same time increasing security, as no devices can overhear the traffic between other devices.
- In connection with specific address processing rules in the switch it also reduces the danger of message injection and message flooding attacks by allowing only certain message flows based on sender and receiver addresses/ports.
- It is possible to verify by visual inspection that no additional network devices are in the loop.

For fault-tolerance there are two switches per bay, one for each of the two redundant protection loops. This redundancy is mainly for safety, not security reasons. It deals with the risk that one of the sensor - protection relay - circuit breaker loops ceases to function, but it is also beneficial for security, as such a malfunction may be caused by an attack. In fact, in section 4 it will be shown that all attacks can be reduced to this kind of attack.
In Figure 1, one can see a substation automation system which is distributed over $2+n$ distinct locations within the substation perimeter: station, station level busbar protection controller, and $n$ bays, with current and voltage sensors (TCTR/TVTR), circuit breakers (XCBR), disconnectors and earthers (represented by XDIS), as well as distance protection (PDIS), local unit of the busbar protection (PBDF) and bay control device which serves to forward information and operator commands between bay level and station level. Also for fault-tolerance, the backup distance protection is connected to a circuit breaker by means of a direct electrical connection, bypassing any electronic networking devices. The station side ports of the bay control devices of all bays as well as all station level operator work stations are connected to the station switch. The station side ports of all local busbar protection devices are connected with the busbar protection central controller via another Ethernet network centered around the busbar protection switch.

3 Attack Scenario Analysis

3.1 Assumptions

For the security analysis a number of assumptions are made about the system and its environment:
– The configuration of the automation system is static. The number and types of devices in the bay level are well-known and basically constant over time within the system. During operation, the configuration only changes in the context of major maintenance or modification work. This justifies the effort of e.g. statically setting up tables with communication partners_ADDRESSES in all devices involved at the time of installation.
– The SAS is not used for billing and there are no other confidential data on the SAS network.
– There will only be specialized substation automation devices (IEDs) connected to the network on the process (bay) level, but no general purpose PCs. The bay level network and devices are connected to the station level above, where the operator interfaces are located, by means of the bay controller. Therefore the bay controller also acts as application level bridge and thus a kind of firewall for the bay level process bus, as shown in Fig. 1.
– All appropriate technical and administrative means are taken to ensure that only authorized and trustworthy personnel has access to the substation automation and communication equipment [5], as an attacker with physical access to the station can with little effort modify connections, add or remove network devices of his own choosing (e.g. sniffer, switches, bridges) at any point in the network, which would undermine any network security measures. This leaves malicious devices, that is, IEDs which in addition to/instead of their normal functionality execute actions that damage the system, as the main attack vehicle.

3.2 Scenario Analysis Example

Fig. 2 shows one of the three main scenarios for network-based attacks on an SAS, ‘failure to break circuit’. The other two scenarios are ‘unnecessary disconnect’ and ‘operating maintenance switches under load’ are not described in this paper due to lack of space.

In the scenario of Fig. 2 a short circuit in a line of the electric grid occurs but the line is not disconnected as it should happen according to the line protection scheme. This, in consequence, may lead to damage in the primary equipment of the transmission/distribution infrastructure and to power outages on the consumption side. Only the inhibition of the switch off is considered as attack here. The actual short circuit in the line which together with this attacker induced fault leads to damage may or may not be a malicious, artificially induced event.

The hatched boxes at the end of each tree denote the atomic generic attack categories to which all network-based threats to the substation automation system can be reduced:

1. message modification,
2. message injection, and
3. message suppression.
4 Countermeasures

This section describes the three attack categories and derives a communication protocol that allows to detect and counter these attacks.

4.1 Message Modification

Parts of the content of an existing, valid message (with authorized sender and receiver) are modified in transit.

Message modification can easily be detected by the receiver if all messages, or their cryptographic hashes [13], are digitally signed by the sender and the receiver has the signature verification key of all authorized senders, which is easily possible as the small number of possible senders are statically known. After detection, the receiver can reject the tampered message, so that this attack reduces to message suppression.

4.2 Message Injection and Replay

Message injection refers to an attacker sending additional messages which are - at least at that point in time - not intended to be sent by any authorized sender. The actual messages that are injected are either
1. completely new messages that are created by the attacker,
2. original, untampered messages previously sent by authorized senders and
   properly received by authorized receivers that have been captured and are
   now resent (replayed) by the attacker, or
3. original, untampered messages previously sent by an authorized sender that
   were intercepted by the attacker and are now, after a delay, forwarded by
   the attacker

These three types of injected messages have different characteristics: The
receiver can detect type 1 messages if a digital signature scheme for the messages
is used, because the attacker will not be able to create a message that carries a
proper signature of an authorized sender. The detection of type 2 and 3 messages
is more difficult, as they are perfectly valid messages which are just sent at the
wrong time (e.g. a scenario 2 attack can be launched by an attacker who has
captured and stored a previous, valid data packet containing a command to open
a circuit breaker). A type 2 message injection can be prevented by the receiver
if the signed message, in addition to the payload data, also contains a sequence
number. Replayed messages will not have the expected sequence number and can
thus be detected and discarded. Care has to be taken that the sequence number
range is large enough to make waiting for range overflow and recurrence of
previous numbers impractical. In particular, the sequence number range should
not be a multiple of a message series period. While the system should tolerate
skipping some sequence numbers to be able to achieve resynchronization after
a message loss, the window of acceptable future sequence numbers should be
restricted, otherwise the system will be vulnerable to replay attacks after the
first sequence number overflow. A timestamp of sufficiently fine granularity (tick
duration smaller than the smallest possible distance between two valid messages)
can also be used as sequence number. A delayed valid message (type 3) cannot
directly be recognized by the receiver. Detection of delayed messages is based
on detecting the non-arrival of the message at the original time. The protocol
for that is described in the next subsection. A delay that is smaller than the
threshold of the message suppression detection protocol cannot be recognized
by the receiver. The system should thus be designed on application level in a
way that messages delayed within the suppression detection time window cannot
cause any damage.

4.3 Message Suppression

Certain messages exchanged between automation devices are prevented from
reaching the receiver. This can either be an attack in itself, e.g. if the circuit
breaker control devices are thus isolated from the protection devices, or message
suppression is used in the context of an attack which injects messages with
malicious content. A timing/delay attack is basically a combination of message
suppression and message injection.

Technically, message suppression can be achieved by various means, e.g.

- reconfiguring a switch or router
- adding a (malicious) gateway to a network segment which drops certain messages
- cutting the wire
- injecting messages (jamming) to congest the network so that the real messages cannot get through

Message suppression cannot be prevented. It can, however, be detected by both communication partners through a combination of heartbeat messages, message sequence numbers, and digitally signed messages. This allows to alert operators to the fact of malicious interference and perhaps even trigger suitable default operations in the receivers (IEDs) involved.

As described above, a suppressed message can be detected by the receiver as soon as a message with a wrong, that is, too high sequence number arrives. This procedure is sufficient for periodic communications (e.g. transmission of sensor values), but does not allow detection of suppression of event-triggered communication (e.g. commands to the circuit breaker) by the receiver. Event-triggered communication channels can be turned into periodic communication channels by regular messages without payload, so-called heartbeat messages, whose only purpose is to assure the receiver that communication is still possible.

### 4.4 Secure Communication Protocol

The following protocol is a summary of the countermeasures described above. It contains all the elements considered relevant to achieve secure communications for substation automation systems.

- Sender and receiver agree on a starting sequence number in a secure way, e.g. at system configuration and start-up time.
- Only the sender knows its private key. The keys were distributed out-of-band at system installation/configuration time.
- Sender transmits periodically. If no meaningful data are to be transmitted, an empty message may be sent.
- Sender adds sequence number to the part of the message that is to be signed.
- Sender signs message with his private key.
- Sender increases sequence number.
- Sender transmits signed message.
- Receiver has the public key of the sender. The keys were distributed out-of-band at system installation/configuration time.
- Receiver verifies the sender’s signature using the sender’s public key. The message must contain some well-known values to allow the receiver to recognize a valid message. This is especially important for data packets that are otherwise purely numerical values.
- Receiver verifies that the sequence number of the message is within the permitted window and increases lower limit of the window, otherwise discards message.
– Receiver knows the period of the regular message and can detect suppression by means of time-out of a timer which is reset each time a valid message arrives.

An attacker can use the sender’s public key to decrypt and read message. He can also change the plaintext message, but without knowledge of sender’s private key he cannot re-sign a modified message in a way that the receiver could again apply the original sender’s public key and obtain a valid message.

For the same reason, it is, in principle, not dangerous that the attacker can observe the sequence number of any message sent. However, encrypting the message sequence number provides additional security in the case of systems with frequently overflowing sequence numbers.

Encryption and digital signatures may lead to non-negligible additional processing capacity requirements and delays [1]. Further studies are necessary in this area.

Care has to be taken that the keys are appropriately protected - private keys against disclosure, public keys against modification - while stored inside the networked devices. It is important that the keys are not factory defaults used in different plants but are set specifically for each individual automation system at configuration time. The keys for digital signatures and possibly encryption need to be explicitly distributed, preferably at system configuration time using an out-of-band channel, e.g. manual installation from hand-held key generator equipment. Automated key exchanges are risky due to ‘man in the middle’ attacks and missing means for out-of-band verification. In a variant of the above protocol instead of digital signatures based on asymmetric cryptography, symmetric encryption with secret keys would be used. In this case one secret key needs to be established and distributed for each sender-receiver pair. Symmetric key encryption is considerably faster than private key encryption. In practice, however, the difference may be less significant if the asymmetric keys are only used to establish a symmetric session key. Refer to [15] for more information about algorithms used for digital signatures and possible caveats for their use.

The above abstract protocol is not restricted to Ethernet-based systems and different actual implementations, both domain specific as well as using Internet standards such as IPsec [8], [6], [7] are possible. [3] contains an analysis of IPsec and some guidelines on how to best make use of it.

5 Conclusion

Nowadays realistic scenarios for network-based attacks on substation automation systems, with respect to both motivation and technical feasibility exist. These attack scenarios differ significantly from attacks in the office environment as confidentiality is not the prime issue.

Currently, substation control and protection devices are connected by electrical direct wiring, by a proprietary fieldbus, or by an open networking protocol all of which do not have any security mechanisms built in and are thus equally unsecure.
However, it has been shown that, independent of the networking technology used and the different intrusion points and techniques for attacks the various network-based attack scenarios on substation automation systems can be classified and reduced to three categories: message insertion, modification, and suppression. These can be dealt with using a combination of a security protocol implemented on top of the networking protocol and conventional safety/fault-tolerance mechanisms.

Major challenges remaining are on the one hand the implementation of these security protocols into the SAS while sustaining the necessary processing and communication performance and on the other hand securing remote access to the SAS via public networks.

References

A Bayesian Belief Network for Reliability Assessment

Bjørn Axel Gran¹ and Atte Helminen²

¹ OECD Halden Reactor Project, P.O.Box 173, N-1751 Halden, Norway
bjorn.axel.gran@hrp.no

² VTT Automation, P.O.Box 1301, FIN-02044 VTT, Finland
atte.helminen@vtt.fi

Abstract. The objective of this paper is to present work on how a Bayesian Belief Network for a software safety standard, can be merged with a BBN on the reliability estimation of software based digital systems. The results on applying BBN methodology with a software safety standard is based upon previous research by the Halden Project, while the results on the reliability estimation is based on a Master’s Thesis by Helminen. The research is also a part in the more long-term activity by the Halden Reactor Project on the use of BBNs as support for safety assessment of programmable systems. In this report it is discussed how the two approaches can be merged together into one Bayesian Network, and the problems with merging are pinpointed.

1 Introduction

With the increased focus on risk based regulation of nuclear power plants, and in accordance with the new generic guideline for programmable safety related systems, IEC-61508, [1], probabilistic safety integrity levels are given as requirements for safe operation. Therefore, there is a need to establish methods to assess the reliability of programmable systems, including the software. Risk assessment based on disparate evidences using Bayesian Belief Net (BBN) methodology is an ongoing activity by the Halden Project (HRP), [2]. Similar studies on the reliability of software-based systems using Bayesian networks have been made within the VTT Automation (VTT), [3].

One objective of this co-operative project is to investigate how a network, representing the software safety guideline and different quality aspects, developed by HRP, [4], can be merged with a network, developed by VTT, representing evidence from disparate operational environments. We also wanted to investigate how easy it is to merge together two different networks build with different tools. HRP have used HUGIN, [5], and SERENE, [6], in their modelling, which are mainly based on conditional probability tables (cpt). VTT have applied WinBUGS, [7], which is based on continuous and discrete distributions and sampling from these distributions. Finally, possible applicability of the merged network is discussed and topics for further investigation are pinpointed.
Reliability estimation method based on BBN is a flexible way of combining information of disparate nature. The information may vary from quantitative observations to human judgments. The objective of using BBNs in software safety assessment is to show the link between observable properties and the confidence one can have in a system. The theory about BBNs is well established, and the method has been applied with success in various areas, including medical diagnosis and geological exploration. There has also been an activity to apply this method for safety assessment of programmable systems, see the European projects SERENE [6] and IMPRESS [8], as well as works by Fenton, Neil and Littlewood, [9, 10, 11]. A description of Bayesian interference, Bayesian Network methodology and theory for calculations on BBNs can be found in books by Gelman et al., [12], Cowell et al., [13], a report by Pulkkinen and Holmberg, [14], and older references such as Whittaker, [15], and Speigelhalter et al., [16].

2 The Halden Project Approach

2.1 M-ADS and DO-178B

The research described in this section was done in an experimental project carried out by a consortium composed of Kongsberg Defence & Aerospace AS (KDA), Det Norske Veritas, and HRP. First of all the project goal was to evaluate the use of BBN to investigate the implementation of the DO-178B, [17], standard for software approval in the commercials world. To reach that objectives a computerized system for automated transmission of graphical position information from helicopters to land based control stations, (M-ADS), developed by KDA, was selected and studied, [4]. The work described below uses parts of the M-ADS system to exemplify the software development process according to DO-178B standard. Please note that references to the system developed by KDA, and evaluated in the project, represent by no mean any official policy of KDA.

The purpose of the DO-178B standard is to provide a required guideline for the production of safety critical software for airborne systems. This guideline was chosen for the study since the M-ADS system is applied in civil aviation, and was previously qualified on the basis of this standard. The main recommendations in DO-178B are given in a set of 10 tables. Each table relates to a certain stage in the development and validation process, and contains a set of objectives.

2.2 The Higher Level BBN

The M-ADS evaluation consisted of several tasks. The first was to construct BBNs on the basis of DO-178B. The BBN was constructed in two levels: higher and lower. The higher-level network consists of two parts: the "quality-part" (or soft-evidence part), and the "testing-part", as indicated in fig 1. Remark that the network was presented a
little different in [2], although that the context is the same. The "quality-part" represents four quality aspects.

- **Quality of the producer**: including the reputation and experience of the producer, quality assurance policy, quality of staff etc.,
- **Quality of the production process**: a high quality implies that the system is developed according to guidelines for good software engineering, that all phases are well documented, and that the documentation shows that the system at all development phases possesses desirable quality attributes as completeness, consistency, traceability etc.,
- **Quality of the product**: including quality attributes for the final product, as reliability, simplicity, verifiability etc., and
- **Quality of the analysis**: including all activities performed to validate the correctness of the system during all stages of the system development.

This part leads to an end node “N-hypothetical”. The intention is to express that the information in the upper network is equivalent to that the system is tested with N randomly chosen inputs without failure. The computation of the "quality-part" of the BBN is based on observations in the lower level networks, and cpts to the edges in the BBN.

The "testing-part" represented by the node "Y: failures in N new tests", describes the connection between hard evidences, Y=0 failures in N tests, and the failure probability of the system (in the context, usage, environment, etc. the system is tested). The failure probability can be interpreted either as a number of failures on a defined number of demands, or as a number of failures on a defined time period. For the defined number of demands N with the constant failure probability p the random number of failures Y has a binomial distribution.

![Bayesian Belief Network](image)

**Fig. 1.** An updated version of the higher-level network, the nodes grouped by "...") represent the "quality-part", and the nodes grouped by "---" represent the "testing part".
2.3 The Lower Level BBN and the M-ADS Evaluation

The lower level BBNs were constructed by applying the quality aspects as top-nodes in four BBNs. Each top node was linked to intermediate nodes representing the 10 lifecycle processes presented in DO-178B. Each of these nodes was again linked to other intermediate nodes, representing the objectives of the tables. The further step was to identify a list of questions to each objective. These questions were based on the understanding of the text in the main part of DO-178B, and then in general formulated so that the answer could be given by a "yes" or a "no".

The elicitation of conditional probability tables (cpt) to the nodes and edges was done as "brain-storming" exercises by all project participants, based on general knowledge and experience in software development and evaluation. Finally all this information together with observations from the system development (KDA) were fed into the HUGIN and SERENE tools, to make a variety of computations, with the aim to investigate different aspects of the methodology, [6]:

- What is the effect of observations during only one lifecycle process?
- How does the result change by subsequent inclusion of observations?
- How sensitive is the result to changes in individual observations?

3 The VTT Approach

3.1 Combining Evidence

The main sources of reliability evidence in the case of safety critical systems considered in the VTT approach are depicted in fig 2, [10].

![Fig. 2. Main sources of reliability evidence in a case of safety critical system.](image-url)

Part of the evidence may be directly measurable statistical evidence, such as the evidence obtained through operational experience and testing. Part of the evidence may be qualitative characterization of the system such as the design features and the development process of the system.
The qualitative characterization of the design features and the development process follows certain quality assurance and quality control principles, which are based on applicable standards. The more strict standards the characterizations fulfil the more reliable the system is believed to be. The evidence based on qualitative characterization can be considered as soft evidence, while evidence obtained from operational experience and testing can be considered as hard evidence. The exploitation of soft evidence in the reliability analysis of software-based system requires extensive use of expert judgment making it quite an unforeseeable matter and therefore the VTT approach is mainly focused to the utilization of hard evidence.

The reliability of a software-based system is modelled as a failure probability parameter, which reflects the probability that the automation system does not operate when demanded. Information for the estimation of the failure probability parameter can be obtained from the disparate sources of hard and soft evidence. To obtain the best possible estimate for the failure probability parameter of the target system all evidence should be combined. In this approach this combining is carried out using Bayesian Networks. The principle idea of the estimation method is to build a priori estimate for the failure probability parameter of software-based system using the soft and hard evidence obtained from the system development process, pre-testing and evaluating system design features while system is produced, but before it is deployed. The prior estimation is then updated to a posterior estimate using the hard evidence obtained from testing after the system is deployed and from operational experience while the system is operational. The difference between disparate evidence sources can be taken care in the structural modelling of the Bayesian Network model.

To analyse the applicability of Bayesian Networks to the reliability estimation of software-based systems we build Bayesian Network models for safety critical systems. The different models are distinguished by the evidence, which is collected from different systems and from different operational profiles. The system and operational profile configurations under consideration are characterized in Model 1 and Model 2. The modelling is done using the WinBUGS program.

3.2 Model 1: Evidence from One System with One Operational Profile

The Bayesian Network shown in the left part of fig. 3 describes a system, for which the observed number of failures $Y$ is binomial distributed with parameters $N$ and $P$. Parameter $N$ describes the number of demands in the single test cycle and parameter $P$ is the random failure probability parameter. This model can be further extended to represent a system with several test cycles using the same operational profile.

To increase the flexibility of the model depicted in the left part, we include a logit-transformed $P$ parameter $\Theta$ into the network, and the network becomes as shown to right in fig 3. The Bayesian network represented as Model 1 can be used in the reliability estimation of a software-based system attached with binomial distributed hard evidence under unchanged operational profile.
3.3 Model 2, Evidence from One System with Two Operational Profiles

The hard evidence obtained for the reliability estimation of software-based systems is usually obtained from both, testing and operational experience. If the testing has been carried out under the same operational profile as the operational experience, the Bayesian Network becomes same as the network shown in fig. 3. Often this is not the case, and the system is tested with a different operational profile under different operational environment. Since the errors in the software are triggered causing a loss of safety function only when certain input occurs, the different operational profiles provide different failure probabilities for the same system. However, the failure probability from testing gives us some information about the failure probability of the system functioning in a different operational profile than where the testing was made. The evidence provided by testing is very valuable and we should make a good use of it by taking into account the difference in the operational profiles when building the model.

The problem of different operational profiles is solved by first connecting the binomial distributed evidence from different operational profiles to separate failure probability parameters, and then the logit-transformed failure probability parameters are connected to equal each other. The difference in the operational profile of the two failure probability parameters is carried to the model by adding a normal distributed random term $\Omega^*$ to the logit-transformed failure probability parameter obtained from testing. The parameters of the normally distributed random term correspond to our belief of the difference between the two operational profiles. The Bayesian Network representing the case is illustrated in fig. 4 when considering only the upper layer.

The parameters connected to the evidence obtained from the testing are illustrated by parameter names with stars. The fundamental idea behind the parameters $\mu^*$ and $\sigma^*$, is discussed in the Master Thesis by Helminen, [3].

4 Merging the HRP Approach and the VTT Approach

Merging the two networks is based on a simplified version of the network presented in fig. 1 and the network shown in fig. 3 and the merged network is described in
The merging was done by starting with the "quality part" of the BBN in the HRP approach. First the node representing restrictions on the hypothetical N was removed. That means, we assumed a direct dependency between the node “N-hypothetical” and the node P. The next step was to replace the node "N-hypothetical" by the node $\Theta_{\text{prior}}$. This was done by transformation of the cpt for $P(\text{N-hypothetical}|\text{Quality of product, quality of analysis, solution complexity})$ into continuous normal distributions. Each of the quality aspect nodes is connected to quality aspects, as described in section 2.2. That allowed us to directly insert the observations from the M-ADS evaluation in the network, and for the merged network we performed calculations for two different scenarios:

Fig. 5. A merged network
1. For were we have no M-ADS observations, but zero failures (Y=0), running from N=100 to N=1000000.

2. For were we have the M-ADS observations, and zero failures (Y=0), running from N=100 to N=1000000.

For both scenarios the calculations were done both by applying HUGIN/SERENE and WinBUGS, and the target was the node for the failure probability. The reason for performing calculations by applying both tools is that while HUGIN gives good support for calculations with conditional probability tables, WinBUGS gives good support for continuous distributions.

In fig. 6 and fig. 7 both the median and the 97.5% percentile posterior distribution values for P on the logarithmic scale are shown. The values for N=1, are the values representing the prior distributions, i.e. before starting the testing (and observing Y=0). Remark that the curves for the 97.5% percentiles are somewhat "bumpy". This due to the fact, that the values are deduced from the posterior histograms.

The first observation is found by comparing the two figures. One sees that the results computed by HUGIN and WinBUGS give approximately the same results.

![Graph showing median and 97.5% percentile posterior distribution values for P on the logarithmic scale.](image)

**Fig. 6.** Median and 97.5% percentile posterior distribution values for P on the logarithmic scale, for the scenario of no KDA observations and the scenario with the KDA observations, calculated by applying HUGIN/SERENE.

The next observations are found by evaluating the different graphs. The results show for a low number of N, e.g. 100, that the posterior failure probability P is lower with inserting the M-ADS-observations, than performing the calculations without any "soft evidences". For a higher number of N the weight on the prior distribution based on the M-ADS observations is reduced, and the two scenarios converge against the same values. We observe that the two scenarios converge for approximately N=10000. This is in accordance with the posterior distribution of $\Theta_{\text{prior}}$ ("N
hypothetical") after inserted the M-ADS observations. This posterior distribution is also a result of the topology (cpts and networks) given by expert judgement in the M-ADS project. [2].

![Graph showing median and 97.5% percentile posterior distribution values for P of the logarithmic scale, for the scenario of no KDA observations and the scenario with the KDA observations, calculated by applying WinBUGS.](image)

**Fig. 7.** Median and 97.5% percentile posterior distribution values for P of the logarithmic scale, for the scenario of no KDA observations and the scenario with the KDA observations, calculated by applying WinBUGS.

The same effect by reducing the expected value of $\Theta$, and a convergence towards $P=0.00001$ is also in accordance with the results presented by Helminen in his master thesis, [3]. However, a direct comparison between the results here and the results presented by Helminen is more difficult. The reason is that the priori distribution of $\Theta$, transformed from the cpts in the HRP approach, is not continuous normal distributed, and it has a larger standard deviation than the scenarios presented by Helminen, [3]. There might also be some divergences in the results presented here due to the approximation of the Binomial distribution by the Poisson distribution in the calculations in HUGIN/SERENE.

## 5 Further Work

The main differences between the two studies lie in the difference of focus areas. The work by VTT mainly focuses to studying explicitly the influence of prior distributions to the reliability estimation and to the investigation of combining statistical evidence from disparate operational environments. The work by the HRP has mainly focused on how to model a software safety guideline, DO-178B, [18], and how to combine "soft" evidences in the safety assessment of a programmable system, [6]. The key idea is to split the larger entities of soft evidence into smaller quantities. Another
difference is the comprehensive usage of continuous distributions in the VTT work, which is somewhat a different approach than the approach used in the HRP study.

The merged networks show how the two approaches can be merged. It gives an extended description of the quality aspects, originally modelled by the node $\Theta$ in the VTT approach, and it shows how different operational profiles, can be included in the approach from HRP. This means that multiple operational profiles may be introduced to the model, in addition to the sources described in model 2. Observations from the testing and operational experience evidence of different power plants using the same software-base digital system under different operational and environmental conditions can also be included. This can e.g. point out a possibility for how to apply BBN in the assessment of COTS (Commercial Off the Self Software). Calculations with evidence representing disparate operational environments, and evaluating possibilities for sensitivity analysis will also both be addressed in the future, as the work presented in this paper is part of a long term activity on the use of BBN’s.

Acknowledgement

We want to acknowledge O. Ventä, U. Pulkkinen, P. Haapanen, VTT, and G. Dahll, HRP, for taking part in the discussions on the applicability of BBN's for reliability assessment. We also want to acknowledge the project team that performed the M-ADS project and Hugin Expert A/S for allowing Gran the use of the HUGIN for his Ph.D.

References

5 Tool made by Hugin Expert a/s, Aalborg, Denmark (http://www.hugin.dk).
Checking General Safety Criteria on UML Statecharts

Zsigmond Pap, István Majzik and András Pataricza

Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
H-1521 Budapest, Műegyetem rkp. 9.
[papzs,majzik,pataric]@mit.bme.hu

Abstract. This paper describes methods and tools for automated safety analysis of UML statechart specifications. The general safety criteria described in the literature are reviewed and automated analysis techniques are proposed. The techniques based on OCL expressions and graph transformations are detailed and their limitations are discussed. To speed up the checker methods, a reduced form for UML statecharts is introduced. Using this form, the correctness and completeness of some checker methods can be proven. An example illustrates the application of the tools developed so far.

1 Introduction

As the complexity of safety-critical systems increases, the task of the engineers in specifying the system becomes increasingly difficult. Specification errors like incompleteness and inconsistency may cause deficiencies or even malfunctions leading to accidents. These problems are hard to detect and costly to correct in the late design phases. The use of formal or semi-formal specification and design languages and the corresponding automated checker tools may help the designer to avoid such faults.

Nowadays UML (Unified Modeling Language [1]) is the de-facto standard for the object-oriented design of systems ranging from small controllers to large and complex distributed systems. UML can be used to construct software specification of embedded systems [2], often implementing safety-critical functions. The well-formedness rules of UML (defined in a formal way) helped its spreading in the area of safety-critical systems. Of course, the general syntactic rules of UML are not enough to guarantee the correctness of the specification. UML models are often incomplete, inconsistent and ambiguous. Tool assistance is required to help the designer to validate these properties of the specification.

Our work aims at the elaboration of methods and tools for the checking of the most important aspects of completeness and consistency in UML models. We concentrate especially on the behavioral part of UML, namely the statechart diagrams. It is the most complex view of the specification, which defines the behavior of the system. Sophisticated constructs like hierarchy of states, concurrent regions, priority of transitions etc. help the designer to structure and organize the model, but their careless use may also lead to specification flaws.

1 Supported by the Hungarian Scientific Research Fund under contract OTKA-F030553.
Our examination is focused on embedded control systems. In these systems, the controller continuously interacts with operators and with the plant by receiving sensor signals as \textit{events} and activating actuators by \textit{actions}. UML statechart formalism allows constructing a \textit{state-based model} of the controller, describing both its internal behavior and its reaction to external events.

The paper is structured as follows. Section 2 motivates our work and presents the model we will use in the paper. Section 3 introduces the basics of the safety criteria. In Section 4 the possible checking methods are discussed. Section 5 describes the so-called reduced form of UML statecharts that was defined to help in proving the correctness of the checker methods and also to accelerate the checking of the model. The paper is closed by a short conclusion.

2 Motivation

The work on automated checking of general safety criteria was partially motivated by our experience gathered during the design of Clt4iT, a safety-critical, multi-platform, embedded real-time system: a fire-alarm controller which is a part of a complex fire/gas/security alarm system.

The Clt4iT system consists of a central unit and a set of data units that collect data from the detectors (smoke, fire, gas, etc.). Every unit can handle alarms independently, and has its own user interface. Since the amount of data originating in the units is large (detector information, alarms, logs etc.) and the communication bandwidth is low, only the recently changed data should be read into the central unit. The task of the central software is to keep record of the aging of data, poll the units, read the changed data or unconditionally refresh the aged ones. All units are monitored in this way; the units that are currently displayed on the screen are called active ones.

Fig. 1 presents one of the statechart diagrams of the central unit software. Its responsibility is to handle the data aging for a given group of data. For each group, there is a time limit defined for data being “old” and “very old”, in the case of active operation, non-active operation, “changed” or “unchanged” data.

The above-presented statechart defines the behavior of a “Model”-type class, which belongs to the internal data model of the system. In this type of class, there must be a distinguished state "Unknown" (which is the initial state, and represents that the internal data model is not up-to-date) and time-out transitions from each state to this "Unknown" state.

The original version of the alarm system (which had to be replaced due to the change of the requirements) was developed on the basis of a verbal specification, without the use of UML. The design and implementation required more than half a year. Despite of the careful testing, residual software faults caused regular crashes of the system. The reasons of these failures were not found.

The development of the new version started by a systematic modeling using UML. As a modeling tool, Rational Rose was used which supports XMI model export \cite{9}. In this case the design and implementation required 4 months.

Our goal was to develop automated tools for the checking of the dynamic UML diagrams of the design to highlight specification flaws early in the design process.
3 General Safety Criteria

N. G. Leveson and her colleagues have collected 47 safety criteria for the specification of software systems [3] and also elaborated checker methods for some of the criteria applied to the statechart-based languages RSML and SpecTRM [5, 6, 7]. The criteria are general and can be applied to all systems independently of the application area. In fact, they form a requirement safety checklist. These criteria can be grouped into several categories as follows: state space completeness, input variable completeness, robustness, non-determinism, time- and value limits, outputs, trigger events, transitions and constraints. The most important groups of these criteria target the completeness and consistency of the specification.

Our main goal was to apply and check these existing criteria on UML statechart specifications. (The checking of a full UML model including object-oriented features like inheritance requires developing new criteria, which is out of the scope of this paper.) Accordingly, we had to formalize and adapt the criteria to UML statecharts and elaborate tools for automated analysis. Formalization and adaptation is a crucial task since the criteria are given in natural language thus they cannot be checked directly. Moreover, some criteria must be decomposed into a set of rules in order to be able to interpret them on the model elements of UML.
In a previous paper [13] we formalized the criteria in terms of the UML model elements and presented an approach to check some selected criteria by applying Prolog rules and by manipulation of the UML model repository. Now our analysis covers the full spectrum of the criteria excluding the ones related to timing and suggests efficient and elegant methods to check those of them that are amenable to automated checking.

### 4 Overview of Checking Methods

The analysis of the criteria proved that more than three-quarters of the criteria can be checked automatically (Fig. 2). Moreover, almost two-thirds of them are static criteria that do not require reachability-related analysis. The criteria that cannot be checked automatically refer mainly to assumptions related to the environment of the system to be checked e.g. environmental capacity, load assumptions, stability of the control loop. They are included in a checklist for manual examination.

In the following, we examine four potential methods for automated checking of the criteria: (1) formalizing rules as expressions of the Object Constraint Language, (OCL [1]) as part of UML, (2) examining the satisfaction of the criteria by graph transformation, (3) executing a specialized checker program and (4) performing reachability analysis. Of course, some criteria can be checked in more than one way; in this case the most efficient one has to be selected. In Fig. 3, three numbers are assigned to each method. The first one gives the number of criteria that can be checked solely by that method. The second one shows how many criteria can be checked theoretically by that method. Finally, the third number shows how many criteria can be completely proven by that method.

In the following, we give an overview of these methods and the typical criteria that can be checked. A more detailed analysis is found in the Appendix and in [16].
4.1 Completeness and Consistency Rules in OCL

The most natural way to express criteria in UML is the application of the Object Constraint Language (OCL), since it is the language that was developed to specify the well-formedness rules of UML. These rules were given by a set of structural constraints interpreted on the metamodel elements of UML [1].

In our case, some of the criteria can be formalized in a similar way, by assigning constraints to metamodel elements. Let us present an example. One of the safety rules requires that all states must have incoming transitions (including the initial transition). Considering only simple states (that have no sub-states), this rule refers to the UML metamodel element SimpleState, and results in the following formalization:

\[
\text{self}\rightarrow\text{forall}(s:\text{SimpleState} \mid s.\text{incoming}\rightarrow\text{size} > 0)
\]

Note that OCL expressions are also well usable to formalize application-specific constraints, e.g. pre- or post conditions.

Constraints interpreted on the UML metamodel can be enforced by a CASE tool that supports the modification of the metamodel. On the other hand, constraints interpreted over the model elements require a common OCL interpreter. In both cases, the checking requires an unfolded statechart model in which the hierarchy and concurrency are resolved, since OCL is not capable of browsing the state hierarchy.

4.2 Graph Transformation Techniques

UML statecharts can be considered as a graph language [11]. Accordingly, graph transformation rules can be defined to modify or transform the statechart models [14]. These transformation rules can be utilized in two ways:

− The model can be transformed into a form that is more suitable for checking. E.g. a hierarchic model can be flattened to check OCL expressions.
− Systematically removing the complete and consistent parts of the model eventually results in the current specification flaws.

Let us consider the following criterion: For all states, there must be a time-out transition defined. It can be checked in the following way:

1. Converting the state hierarchy into a flat model (for details see Section 5 and [11]). The approach is illustrated in Fig. 4.

![Fig. 4. Example for resolving the state hierarchy](image-url)
2. Looking for the following situation: There is a SimpleState in the graph AND there is NO Transition connected to this with the stereotype “TimeOut” OR with an action “OnTimer”.

In general, the graph transformation rules are defined by giving a left side (condition) and a right side (result). The system tries to match the left side on the source model. If it matches then transforms this part of the model into the right side of the rule. The transformation is ready, when the rule does not match any more. In our case, it is not practical to modify the source model. Instead of this, a second model is built, that will contain the result of the transformation steps. Accordingly, the left and right sides of the rule are duplicated, describing the condition and the result including the patterns both in the source and in the target model (of course, the source will not change) [14].

Currently the graph transformations are implemented in Prolog. The UML CASE tool saves the model in standard XMI format (using XML as model description language [15]). This common representation is parsed and loaded into memory as a set of predicates. The rules are executed and the resulting model is saved again in XMI format, which can be loaded into the CASE tool to highlight the specification flaws.

4.3 Checking by Specialized Programs

Some criteria cannot be checked by graph transformation and/or the assignment of OCL constraints. We mention here one criterion: for each state and each trigger event, the guard conditions must be mutually disjoint. The checking of this rule requires the interpretation of the guard expressions, which cannot be done by a general-purpose OCL interpreter (that targets structural constraints) or by graph transformation (as the values of the guards dynamically modify the model structure).

To verify the guard conditions, we restrict the use of guards similarly to RSML [5]. We require expressions built from atomic propositions (that are true or false) connected by Boolean operators OR, AND or NOT. Accordingly, we can assemble a disjunctive normal form of the propositions and a truth table of the terms.

Using this form, the guard conditions can be converted to events and transitions. After a standard optimization, which can find and eliminate uniform cases [4], the checker removes all original transitions starting from the given state and triggered by the given event. Then for each term of the normal form (combination of guard condi-

![Fig. 5. Example with two guard conditions](image-url)
tions), it generates a new virtual event and a transition triggered by that virtual event. In this way, guarded transitions will be resolved by virtual events, and the mutual exclusiveness checking is traced back to the checking of the trigger events.

Fig. 5 (a) shows one state from our example. According to the guard expressions "IsChanged" and "!IsChanged", here two virtual events are generated from the original event "NewData" (Fig. 5 (b)). The original transition is replaced with the ones triggered by the virtual events (Fig. 5 (c)).

4.4 Reachability Analysis

There are criteria that require reachability analysis. To formalize and check these criteria, temporal logic expressions and model checking can be used.

Typical reachability problem is the checking of the existence of unreachable states and transitions that can never fire. The rule that prescribes that each output action must be reversible is a similar problem. Another important consistency criterion is related to the avoidance of nondeterminism. In UML statecharts, one source of nondeterminism is the simultaneous firing of transitions in concurrent regions of composite states. In this case the order of their actions is not defined. The suspicious state pairs can be found by static checking, i.e. looking for situations where transitions in concurrent regions are triggered by the same event, guards can be true at the same time and there are actions defined. However, the static checking cannot claim that these state pairs are reachable during the execution of the system.

We use the model checker SPIN [10] as external tool to decide reachability problems. The UML statechart is transformed to Promela, input language of SPIN [8], and the reachability problem is formulated in linear temporal logic (LTL).

5 The Reduced Form of UML Statecharts

During the elaboration of the checker methods and identification of the basic rules that are sufficient and necessary to check the criteria, we discovered that checking of several criteria could be traced back to the same set of basic steps. The common characteristic of these steps is that their execution results in a simplified, flattened model structure that is easier to check (both by OCL constraints and graph transformation). We call this model structure the reduced form of the statechart.

The reduced form was utilized also during the formal proof of the correctness and completeness of the proposed checking methods. For a given criterion, it is proved first that the steps generating the reduced form preserve the properties to be checked. Then the proof of the later steps can be built on the relatively simple and formally tractable reduced form.

Fig. 6 shows the UML metamodel of the reduced form of statecharts. It has several advantages. The special UML elements are removed or converted into the set of "standard" ones consisting of simple states, transitions, events and actions. The hierarchy is resolved and the model is fully static, no guard conditions are in the model.

The reduced form is generated by a series of graph transformation steps as follows:
1. Multiple statechart diagrams in the UML model are merged into a single diagram.
2. Associations are inserted among states and events (the checker must verify all states and all possible events on that state, i.e. the Cartesian product of the set of SimpleStates and Events).
3. Temporary states (SimpleStates that have completion transitions, i.e. an output transition without a trigger event defined) are eliminated, since they are not part of any stable state configuration. The completion transitions are converted into a set of regular transitions, where there is exactly one transition for each possible event – this method also saves the information of the guard conditions.
4. Associations are inserted between each pair of concurrent states. Since the state hierarchy will be converted into a flat model, the information on the concurrency of states should be kept.
5. The state hierarchy is converted into a flat model. Every SimpleState inherits the outgoing transitions of its parent states and the initial states inherit the incoming transitions of their parent states. The associations between the SimpleStates and their parents are preserved.
6. Entry (exit) actions are moved to the incoming (outgoing) transitions. Entry (exit) actions are last (first) ones in the sequence of actions executed by the incoming (outgoing) transitions [12].
7. Internal events are converted into self-loop transitions. Since the entry and exit actions were already removed in the previous step, this step does not violate the semantic rules of UML.
8. Pseudo-states (e.g. initial and final states) and composite transitions are converted into normal states and transitions. Fork transitions are marked, otherwise the resulting transitions starting from the same state and triggered by the same event would result in inconsistency. In the case of join and Sync transitions, the source states are assigned a self-loop transition guarded with an "in_state" condition.
9. Guard conditions are converted into events (see Section 4.3).

Let us present an example how the reduced form is used. Since there are only simple states and transitions in the model of reduced form, the criterion of the completeness of state transitions can be formalized in OCL as follows:
self -> forAll(s:State | s.myevent -> forAll(e:Event | s.outgoing -> select(t:Transition | t.trigger = e) -> size > 0))

Almost all criteria can be checked on the reduced form. In some cases, however, it turns to be more practical to use the original model. E.g. the Promela code used during reachability analysis is generated on the basis of the original statechart.

6 Conclusion

This paper presented methods and tools for the checking of UML statechart specifications of embedded controllers. The existing criteria given in [3] were adapted to UML statecharts and efficient methods were proposed for the automated checking.

The developed methods were successfully applied in the case of the Clt4iT system. The general safety criteria were checked for all statechart diagrams. The automatic checking of a statechart using the graph transformation framework required about 30 seconds in average. Since there was only limited concurrency in the system, the state space explosion problem was practically avoided. By the automated checking, (besides some typing errors) typically incompleteness due to malformed guard conditions and missing transitions were detected in the early design phase. The validation testing detected additionally some non-suitable settings related to timing (that could not be checked in the design phase). The problems that occurred in the previous system did not appear in the new version.

References

12 M. Gogolla: Graph Transformation on the UML Metamodel, Workshop on Graph Transformation and Visual Modeling Techniques, ICALP'2000, Geneva, Switzerland, 2000
Appendix

Table 1 presents the groups of general safety criteria (without the timing related ones) and the methods required to check them. (We introduced groups here because the methods of checking the criteria cannot be clearly separated, and some criteria must be decomposed into several rules.) In the Table, "Yes" means that the method is applicable and necessary, "No" means that the method is not applicable. "-" means that the method is applicable but not optimal to check the given group of criteria.

Table 1. Groups of criteria (without the timing related ones) and the checker methods

<table>
<thead>
<tr>
<th>Group of criteria</th>
<th>Static Methods</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OCL</td>
<td>Graph Transformation</td>
</tr>
<tr>
<td>The system should start in a safe state</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>The internal model must be valid</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>All variables must be initialized</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>The specification must be complete</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>The specification must be deterministic</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>There must be timeout transitions defined</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No path to critical states should be included</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>There must be behavior specified in the case of overloading</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>All states must be reachable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>All valid output values must be generated</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Paths between safe and unsafe states (soft and hard failure modes)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Repeatable actions in live control loops</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Limits of data transfer rate must be specified</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>The time in critical states must be minimized</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>The output actions must be reversible</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>All input information must be used</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Control loops must be live and checked</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>All input values must be checked</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>The overloading parameters must be defined</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Presenting a Safety Case
– A Case Study –

Odd Nordland

SINTEF Telecom and Informatics
Systems Engineering and Telematics
NO-7465 Trondheim
Odd.Nordland@informatics.sintef.no
http://www.informatics.sintef.no/~nordland

Abstract. A brief description of the Computer Based Interlocking system that is to be introduced in Norway is given and the requirements for presenting a safety case are described. Problems in actually fulfilling those requirements and lessons to be learnt are explained.

1 Introduction

Automatic train protection (ATP) and interlocking systems have become more complex and rely heavily on microprocessors and software. Thus, the assessment and certification of such new systems has changed and become correspondingly more complex. In addition, European integration leads to a need for common principles and procedures, so that assessments and certifications performed in one country can be adopted by other countries.

One step in this direction is the adoption of European standards for railway applications, notably the CENELEC (pre-) standards EN 50126 [1], EN 50128 [2] and prEN 50129 [3]. They describe processes to be followed in order to be able to assure the safety of a railway application. However, whilst they describe reasonably completely what to do, they do not go into great detail on how to do it. Ref. [1] is the top-level document that covers the overall process for the total railway system. It defines Safety Integrity Levels and sets the frame for the more detailed activities that are described in ref. [2] and ref. [3]. The latter is the standard that defines the activities for developers and manufacturers, but also describes the requirements that a third party assessor shall verify. Ref. [2] is the software specific "subset" of ref. [3].

In this paper, an actual application of the standards to a computer based interlocking system and some of the problems that were encountered will be described. The lessons learnt in the process will help to make future certifications more efficient for all involved parties.
2 The Computer Based Interlocking System

The Norwegian railway administration Jernbaneverket (JVB) signed a framework agreement with Adtranz Norway for delivery of a number of EBILOCK 950 systems to the Norwegian market. Adtranz Sweden produced the EBILOCK 950 system for a worldwide market, so an adaptation of the system to the Norwegian market was necessary.

In the course of the project, the system concept was modified, and the product is now referred to as Computer Based Interlocking "CBI950". The adaptation to the Norwegian market is referred to as CBI950_JBV. CBI950 is a development based on an older system that has been in use for many years. That system was designed and built long before the CENELEC standards came, so the processes followed and the documentation produced were not compliant with what the standards require today.

CBI950 is a platform consisting of hardware, software and a process for generating application specific software. The hardware consists basically of an Interlocking Processing Unit (IPU) and an Object Controller System (OCS). The IPU is a computer system for processing the interlocking rules and controlling the OCS. The OCS receives orders from the IPU for the control of wayside objects (e.g. signals, point switches etc.), sends corresponding messages to the wayside objects and receives status information from them. The status information is reported back to the IPU for processing there.

The software in the system must be generated for each specific application. It contains generic information about the interlocking rules to be applied and about the kinds of objects that can be controlled, and specific information about the actually connected objects. The generation process makes extensive use of automatic tools in order to achieve the required safety integrity level for the software.

2.1 Adaptation to the Norwegian Market

Adaptation of the generic CBI950 to the Norwegian market affected both hardware and software. For example, the hardware had to tolerate the environmental conditions in Norway, which include temperatures below -30°C and installations at altitudes over 1000 metres above sea level. The software had to be adapted to process the Norwegian interlocking rules and to control the wayside objects used in Norway. User manuals and installation instructions had to be adapted to the modified hardware and software and be translated into Norwegian.

2.2 Applying the CENELEC Standards

In addition, the framework agreement required adherence to the CENELEC standards, so many technical documents had to be updated or even newly generated. Fitting the documentation to the requirements for a safety case became a major task in the project in order to ensure that the Norwegian railway inspectorate would be able
to approve JBV's use of the system! Over five hundred documents have been produced and assessed.

The CENELEC standards were pre-standards when the project started in 1997, and indeed, at the time of writing ref. [3] is still a pre-standard. There was little or no experience with using the standards. Indeed, there are still very few projects that are trying to follow the standards from the outset, and those that do usually involve development of a completely new system. The experiences gained in such projects are seldom published (see, however, ref. [4]), so there is little guidance for newcomers.

The requirements for a safety case were – and are – not widely understood. Before looking at the problems that arose in the process of producing a safety case, we should first look at the requirements for such a safety case.

3 The Requirements for a Safety Case

Ref. [3] requires that a safety case shall be submitted by a manufacturer and assessed by an independent third party before the safety authorities should approve commissioning the system. The term "Safety Case" is perhaps straightforward enough for people with English as their mother tongue, but experience shows a large degree of confusion when non English speaking Europeans use the expression!

The term "case" is used in a variety of contexts. We have uppercase and lowercase letters, special cases, briefcases, suitcases, court cases and - safety cases. The latter is derived from the concept of a court case: the prosecutor and defendant both "present their cases" to the court so that the judge can pass a verdict.

Now for our purposes, somebody has to present the case for the safety of a new (or modified) system so that the "judge" - the safety authority - can reach a decision. As in legal proceedings, the "judge" will refer to an expert assessment by an independent party before relying on his own personal impression. (The word "assessor" did, in fact, originally mean "co-judge"!)

One of the most common problems with safety cases is that they are too concise. The standards do allow the use of references rather than submitting large volumes of documentation for approval, but it simply isn't sufficient to just refer to the documents and state that all the information is there, leaving it up to the assessor to read through them all and extract the necessary facts.

The safety case must in itself contain enough information to give a clear impression of the system's safety properties and indicate where the details can be found if this is desired. So the safety case chapters that are defined in ref. [3] must contain descriptive text rather than a more or less complete list of references. In the following sections, these chapters of the safety case are discussed.
3.1 Definition of System

The first chapter in the safety case is the Definition of System. It shall give a complete and detailed description of the system for which the safety case is being presented. Ref. [3] states:

"This shall precisely define or reference the system/subsystem/equipment to which the Safety Case refers, including version numbers and modification status of all requirements, design and application documentation."

This means that the definition of system shall contain:

- A description of what the system is, its functionality and purpose, with references to the requirements' specification and other descriptive documents.
- The product structure. This is more than just a parts list; it's a document that identifies the components of the system and the way they are related to each other and the overall system.
- Descriptions of all interfaces, both external and internal, with references to the corresponding documentation. The interfaces should be traceable to the product structure.
- Information concerning the issues, revisions and dates of all applicable documentation.

3.2 Quality Management Report

This chapter is a report that describes what has been done to ensure that the system has the required quality throughout the entire life cycle. This involves:

- A description of the quality requirements with reference to the corresponding "source" documents. These are more often than not generic, company internal documents, but there can also be laws or regulations that define quality requirements. Such laws and regulations must be identified.
- A description of the quality management system with references to the corresponding plans and procedures. In other words, a description of what one intended to do in order to ensure the necessary quality. This must also contain a description of the project's organisation and identify by name the people in the various positions and their responsibilities and qualifications!
- A description of what actually was done, with references to e.g. audit reports, minutes of meetings and any other documents concerning the performed activities. In addition, any deviations from the plans and procedures shall be described and justified. With deviation we mean any activities that should have been performed according to the plans, but which either were not performed, or for which no documentation or other evidence can be provided.

---

3 For simplicity, the term "system" will be used in a very generic sense that includes subsystem, equipment, product etc.
3.3 Safety Management Report

This is the corresponding report for safety management. As with the quality management report, the safety management report involves:

- A description of the safety requirements with reference to the safety requirements’ specification. The safety requirements’ specification may be a subset of the requirements’ specification rather than a separate document. In this case, the relevant parts of the requirements' specification shall be identified. In addition, there probably will be laws and regulations defining general safety requirements. These must also be identified.
- A description of the safety management system with references to the corresponding plans and procedures. In other words, a description of what one intended to do in order to ensure safety.
- A description of what actually was done, with references to e.g. the hazard log (see ref. [1], paragraph 6.3.3.3 for a detailed description), safety audit reports, test reports, analyses and any other documents concerning the performed activities. As with the Quality Management Report, any deviations from the plans and procedures shall be described and justified.

3.4 Technical Safety Report

This is the chapter where the safety characteristics of the actual system are described. It shall describe which standards and construction or design principles have been applied to achieve safety, and why these are adequate. It will identify the technical properties of the system and show how they have been demonstrated or confirmed by e.g. test records, test and analysis results, verification and validation reports, certificates and so on.

The technical safety report shall also describe how it is ensured that the system does what it is intended to do, and also how it is ensured that the system does not do anything it was not intended to do, even under adverse conditions. This will lead to expressing "safety related application conditions", i.e. conditions which have to be fulfilled if the system is to remain safe.

3.5 Related Safety Cases

If the system's safety relies on the safety of its parts or components, the corresponding safety cases shall be identified here. In such cases, any restrictions or application conditions mentioned in those safety cases shall be recapitulated here.

Note that this also applies when there is a previous version of the system for which a safety case already exists. In this way, producing a safety case for an upgraded system can be considerably simplified. Since the previous version's safety case contains all the relevant information for that version, only the changes for the new version need to be described.
Similarly, by using modules or components for which safety cases already exist, the need for detailed descriptions and records can be reduced to the supplementary information that is necessary for the system at hand.

3.6 Conclusion

This chapter is the plea that recapitulates the evidence presented in the previous chapters and the argumentation for the system's safety. Any restrictions, limitations or other "application conditions" shall be stated here. Ref. [3] states:

"This shall summarise the evidence presented in the previous parts of the Safety Case, and argue that the relevant system/subsystem/equipment is adequately safe, subject to compliance with the specified application conditions."

3.7 Different Kinds of Safety Cases

Ref. [3], paragraph 5.5.1 identifies three different categories for Safety Cases:

- **Generic product Safety Case** (independent of application)
  A generic product can be re-used for different independent applications.

- **Generic application Safety Case** (for a class of application)
  A generic application can be re-used for a class/type of application with common functions.

- **Specific application Safety Case** (for a specific application)
  A specific application is used for only one particular installation.

The underlying idea is that a Generic Product Safety Case (GPSC) will present the safety case for a product, regardless of how it is used, so that it can be deployed in a variety of different safety related applications. The Generic Application Safety Case (GASC) will present the safety case for an application without specifying the actual products to be used as components. It will simply refer to the generic product properties that the components should have.

Unfortunately, this makes the boundary between GPSC and GASC rather fuzzy, because a complex system that can use “generic” components (and therefore has a GASC) can itself be deployed as a component in a greater, more complex system. Then the “generic application” becomes a “generic product” within the more complex application.

For example, a controller for wayside objects could be a generic application that can control a variety of different objects without specifying exactly which “brands” must be used. Using a particular kind of controller in an interlocking system would make it a generic product within that system. However, this should not influence the evidence for the controller’s safety properties.

Finally, the Specific Application Safety Case (SASC) presents the safety properties of a particular combination of products in a given application. It will, of course, draw on the underlying GPSCs and GASC, but in particular, the details of planning and
operation will be relevant here. In fact, ref. [3] prescribes “separate Safety approval... for the application design of the system and for its physical implementation...”, so there must be two SASCs:
- A “SASC - Design” that presents the safety evidence for the theoretical design of the specific application.
- A “SASC - Physical implementation” that presents the safety evidence for “e.g. manufacture, installation, test, and facilities for operation and maintenance”.

Ref. [3] requires the same structure for all the above kinds of safety case, although the contents of the various sections will depend on the kind of safety case that is involved.

4 Problems Encountered

The certification process has taken much longer than originally planned. This was in part due to the fact that the CENELEC standards were poorly understood, so that the need for certain forms of documentary evidence was not always recognised. This lead to delays and discussions about what to produce and in which form it should be produced. This was a learning exercise that cost time and effort. Hopefully, it will be a one-time exercise, so that future certifications will be more efficient.

One of the major problems was the interpretation of the expression "Safety Case"! There is a widespread misconception in the non-English speaking community that this is a collection of safety-related documentation (i.e. a kind of bookcase) rather than a line of argumentation. This meant that the early versions of the safety cases were structured as guidelines through the relevant documentation rather than as an argumentation that was supported by the documents. Now this does make it easier for an assessor to find his way through the literature, but it's not suitable for presentation to an inspectorate that wants a concise argumentation for the system's safety and the assessor's confirmation that the argumentation is valid.

It took several iterations before the safety cases reached a form that could be presented to the inspectorate.

In addition to the problems in understanding the standards, some additional difficulties were encountered. These were mainly due to matters that are not covered by the standards. The most evident ones are described below.

4.1 Legacy Products and Documents

As mentioned above, CBI950 is a development based on a previous version of the system. This meant that the processes that had been followed did not correspond to what the CENELEC standards demand. The standards do not cover this case: they are tuned to development of new systems rather than adaptation of old ones.

The development processes for the previous versions of products within the system did not fully correspond to the processes described in the standards. Obviously, the processes could not be re-enacted in a different way, so documentary evidence that
they were equivalent to what the standards recommend had to be produced. It should be noted here that the standards do not make a particular process or life cycle model mandatory, so the above approach is conformant with the standards. However, producing the necessary documentary evidence in retrospect can be a time consuming task.

The CENELEC standards also require documentary evidence that key personnel are adequately qualified for their tasks. This can be difficult when personnel have changed in the course of time and the relevant information is no longer accessible.

4.2 Dynamic Development

The system concept was modified in the course of the project. This had considerable effects on the extent to which documents that had been produced during the project could still be used. A fair amount of already assessed documentation became obsolete, and associating old documentation with the new structure was not always a straightforward task.

In addition, technical improvements and modifications were implemented in subsystems. The necessary safety documents were often generated as if the affected subsystem had had no relationship to a previous version. This meant that many documents were regenerated instead of simply re-using the previous product versions and their documentation, and justifying the changes. This made for a clear segregation of product versions, but involved a lot of extra documentation and a correspondingly large amount of extra assessment!

4.3 Embedded Processes

The CBI950 concept includes a process for generating the application software. The CENELEC standards do not deal with proving that a process is safe. Processes are regarded as a means to create a product, and the safety of the product must be demonstrated.

In the case of CBI950, the software generation process is fundamental to the generic application. The underlying idea is that a safe process will lead to safe software. This is the philosophy behind ref. [2], because it is recognised that one can't quantify the safety properties of software. However, ref. [2] only identifies "recognised" processes, it does not describe how a "new" process itself should be certified.

The general structure for a safety case can still be used for a process, but whereas defining the system (here: process) is a fairly clear-cut task, providing justification that the process will lead to safe software is more complicated.
5 Lessons Learnt

The previous three subchapters show that the standards don't cover all aspects of a real life application. It is up to the parties that are involved in the certification process, i.e. the supplier, the railway administration and the assessor, to find a solution for those aspects that the standards do not cover. This should be done at the start of the project, so that all parties know who will do what and when.

Modifications along the way are expensive! The standards foresee the possibility of basing a safety case on one or more "related safety cases", so it is better to get the safety case for the original system finished and then upgrade it for modifications, rather than continuously adapting an unfinished safety case to a changing system.

The CENELEC standards are here to stay. Even if ref. [3] has the status "pre-standard" at the time of writing, it will also be adopted some day in the not too distant future. The standards require well-defined documentary evidence for safety, and the kind of documentation that is required is known today. Manufacturers are well advised to start adapting their documentation NOW. This includes not only the documents that already exist, but particularly new documentation that will be produced in the future.

Manufacturers should also be more aware of the need to document history. As long as there are products around that were developed before the standards were adopted, there will be a need to document things from the past that go beyond the usual, legally prescribed archiving requirements.

Finally, a thorough understanding of the standards is imperative. Learning the standards by writing a safety case is ineffective and expensive. The process must be the reverse: understanding the standards is the key to presenting a good safety case.

6 Recommendations

As mentioned in chapter 3, the safety case should contain information on the issues, revisions and dates of all documents. For this, a separate "List of Applicable Documents" ("LAD") should be used. Such a document list must identify all the applicable documents, not just the directly referenced ones, by title, document number etc. and the exact versions to be used. Then it is sufficient to identify the valid issue and revision of the LAD in the safety case - for all other documents reference can be made to the LAD.

Document your work! Many manufacturers have excellent procedures for producing safe and reliable products. Because everybody in the company is forced to follow such procedures, nobody thinks of actually recording how and when they did what. The result is a lack of evidence for the excellent way that they did their job. Recording it may look like a lot of extra paper work, but it's valuable input to the safety case and also a safeguard against potential liability claims years later.

Use a hierarchical product structure. The concept of related safety cases means that you can reuse the safety cases of lower level products in all more complex systems
that use them. And the safety cases of "simple" subproducts will be easier to produce than a single safety case for a large, complex system.

And finally: teach your people the standards at the beginning of the project. If they're supposed to follow the standards from scratch, they should know them in advance. When they understand what the standards require, they will be much more conscious of what they have to do in order to get the safety case right on the first try!

References

1 CENELEC EN 50126:1999
   Railway Applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)
2 CENELEC EN 50128:2001
   Railway Applications - Communications, signalling and processing systems - Software for railway control and protection systems
3 CENELEC prEN 50129:2000
   Railway Applications - Safety related electronic systems for signalling
4 Skogstad, Ø.: Experiences with Safety Case Documentation According to the CENELEC Railway Safety Norms.
Safety Functions versus Control Functions

Hans R. Fankhauser

Bombardier Transportation, Propulsion & Controls Division
SE - 721 73 VÄSTERÅS, Sweden
hans.r.fankhauser@se.transport.bombardier.com

Abstract. It is argued that there need not be any conflict between “Control Functions” and “Safety Functions” as long as “Functionality” and “Safety” are integral parts of the design process and considered on an equal basis at the earliest stage possible in the development. A practical example is given to illustrate this viewpoint. The need to expand and complement the customary set of notions and methodologies is motivated.

1 Introduction

The (provocative) statement, “Safety Functions versus Control Functions”, has a parallel in “Safety versus Availability” recently addressed by this author [1]. There are similarities but also differences, which need to be pointed out. “Control Functions” are necessary to fulfil the required functionality and their proper design has a decisive influence on the achievable “Availability”. Both, the functional requirements and the requirement on “Availability” are usually part of the “Functional Specification” and hence, already known in the tendering phase of a potential project. On a qualitative level the following consideration is applicable. In the past, “Safety” was often said to be in conflict with “Availability”. In the extreme, equipment that is 100% safe will never be in operation. Intuitively, this is (for a person working in the field) difficult to understand as both properties have their origin in the components making up the equipment. It is also true that often in the past “Reliability (Reliability-Availability-Maintainability, RAM)” analysis as well as “Safety” analysis were performed at the end when all the design was finalised, just to confirm that all requirements were met. Equally often, the targets for “Reliability” and “Availability” were met (especially as long as the requirements were moderate), but “Safety” issues were left to be resolved. Of course, in such a situation, where trade-off actions in favour of the “Safety” side have to restore the balance, it is natural to put the blame on “Safety”.

A way to avoid this confrontation is to make both, “Availability” analysis and “Safety” analysis, a natural part of the design process, right from the beginning. Of course, this conclusion is valid even on a quantitative level. However, it is not enough and here the difference becomes evident. In contrast to the requirements named above, the “Functional Specification” does not contain any specific quantitative requirements on “Safety” beyond statements of the general type: the product and/or the operation should be safe. An explanation for this is that none or at least not all of the “Safety” issues are known at that time. The types of hazards
leading to “Safety” issues are dependent both on the particular technical solution adopted to attain the specified functionality and the interaction with the infrastructure. A “Risk Model” is needed to translate the “Tolerable Risk” based on the current values of society to engineering quantities necessary to initiate the design. In Sec. 3 and 6 this aspect will be dealt with in more detail.

We noted above, that besides the technical solution and infrastructure, society also has to be taken into account when performing a “Safety” analysis. In clear text this means that a continuation of the discussion is meaningful only when we focus on a particular application and address an identified hazard. This is the reason why we restrict our attention to a railway application, specifically to the propulsion system containing all the necessary parts that make the train move. The modern control of propulsion systems involves the generation of harmonics in different ways and for normal operation adequate barriers are provided. However, there are situations (faults) resulting in the possibility that harmonic currents propagate into the infrastructure and interfere (Electro Magnetic Interference, EMI) with track circuits. Track circuits are electromechanical devices that control the signals observed by the train driver. A situation where a “red” signal is turned to a “green” one is called a signalling Wrong Side Failure (WSF) and of course, might have adverse consequences. Hence, the identified hazardous events to be addressed are the failure modes leading to the generation of conducted EMI by the propulsion system, which directly or indirectly is capable of causing a signalling Wrong Side Failure.

Before turning to the main subject we pause for a short discussion of an important concept and an interesting aspect in this context. This concerns the concept of “Proof Check Intervals” and the aspect of “Software Barriers” versus “Hardware Barriers”.

1.1 Proof Check Intervals

In complex systems even daily self-tests are not able to reveal all potential failures which might limit the proper functioning of the provided barriers. Therefore, proof checks have to be performed at regular intervals in order to ensure a stated “Hardware Safety Integrity”. The checks usually require the equipment to be taken out of service. Special set-ups, including access to a computer and extensive manual intervention are necessary to perform the task. This is clearly a limitation in “Availability” in favour of “Safety”. However, if this conflict is observed at an early stage, the adverse consequences can be kept to a minimum. By identification (on a subsystem level) of the dominant contributors to the overall Probability of Failure on Demand (PFD), an optimal subdivision can be achieved. Analogue and digital in/out-units most often dominate the list of contributors. As their function is straightforward there is a chance that by early planning the corresponding proof checks can be incorporated in an extended version of an automatic self-test. Practical examples (e.g. Line Interference Monitor, Sec. 5.2) show that this approach is very efficient and results in tremendous savings.

This example demonstrates again the importance of close and early interdisciplinary contacts of all three sides: “Functionality”, “Availability” as well as “Safety”. It puts high requirements on a straightforward communication between the different organisational units involved and last but not least on the individuals.
1.2 Software Barriers versus Hardware Barriers

There is a common opinion that hardware barriers are “better” than software barriers. Of course, there is more historical experience with respect to the performance of the former ones available. However, an important fact often overlooked is the work (mainly manual interventions) involved for proof checking of their proper functioning. The consequence is that the corresponding proof-check interval chosen for hardware barriers will be substantially longer than that for software barriers. Software barrier constituents are easily integrated in automatic daily self-test procedures and hence, their PFD, or equivalently their Mean Fractional Dead Time (MFDT), is very low. Practical examples show that comparable MFDTs differ by a factor of more than 100 in favour of software barriers. This confirms that the reluctance against the introduction of software barriers is of a psychological nature and not based on technical arguments.

2 Terminology

In the context of the definitions, according to the applicable standard [2], the term “equipment under control (EUC)” applies to the propulsion system. The “EUC control system” consists of the Drive Control Units (DCUs) as well as parts of the Train Communication & Control System. Traditionally, the “EUC risks” to be addressed are divided into EE&CS (Electrical Engineering and Control Systems) and Non-EE&CS issues. The signalling Wrong Side Failures (WSFs) belong to the former ones.

In view of the definitions above, the more neutral term, “Initiating Event (IE)”, as used in our Fault Trees and corresponding analysis reports, is to be considered synonymous with the term “hazardous event”.

3 Design Requirements

Usually, acceptable risk levels are expressed in terms of injury (fatalities) or property damages, which is not at all useful as design criteria. Via a “Risk Model” the corresponding requirements have to be translated to measurable engineering quantities in form of limits for currents, electromagnetic fields with associated critical frequency ranges and/or exposure times.

It is essential that these quantities are established and documented for each of the identified hazards at the earliest stage possible. In this way, all the necessary steps can be agreed on and properly planned for all the phases, i.e. from design to final validation.

It is also important to realise that not only the traction system configuration, but even more so the properties of the infrastructure will determine what types of hazard we have to consider. This means, that the question of the tolerability of the corresponding risk levels can only be answered in a particular application. Furthermore, the decision will have to be based on the current values of the concerned society.
4 Applicable Norm and Safety Functions

As a consequence of the discussion above, it can only be decided for a particular application whether the internal protection functions of the EUC control system provide sufficient integrity against a particular type of hazard, whether it should/could be upgraded or whether there is a need for external risk reduction facilities (e.g. ISS/LIM system, see Sec. 5 and 6).

The applicable standard [3] allows for this view and especially for the latter possibility, as long as certain requirements are fulfilled. They are stated in the paragraph below (with the original reference number in bracket):

(7.5.2.1) Where failures of the EUC control system place a demand on one or more E/E/PE or other technology safety-related systems and/or external risk reduction facilities, and where the intention is not to designate the EUC control system as a safety-related system, the following requirements shall apply:

a) the dangerous failure rate claimed for the EUC control system shall be supported by data acquired through one of the following:
   - actual operating experience of the EUC control system in a similar application,
   - a reliability analysis carried out to a recognised procedure,
   - an industry database of reliability of generic equipment; and

b) the dangerous failure rate that can be claimed for the EUC control system shall be not lower than $10^{-5}$ dangerous failures per hour; and

 NOTE 1 The rationale of this requirement is that if the EUC control system is not designated as a safety-related system, then the failure rate that can be claimed for the EUC control system shall not be lower than the higher target failure measure for safety integrity level 1 (which is $10^{-3}$ dangerous failures per hour).

c) all reasonably foreseeable dangerous failure modes of the EUC control system shall be determined and taken into account in developing the specification for the overall safety requirements; and

d) the EUC control system shall be separate and independent from the E/E/PE safety-related systems, other technology safety-related systems and external risk reduction facilities.

 NOTE 2 Providing the safety-related systems have been designed to provide adequate safety integrity, taking into account the normal demand rate from the EUC control system, it will not be necessary to designate the EUC control system as a safety-related system (and, therefore, its functions will not be designated as safety functions within the context of this standard).

In figure 1 this situation is illustrated and further clarified by the following two examples (the bold text points to the appropriate box in Fig. 1):

i) As an INITIATING EVENT (IE) we may have the “Repeated Operation of a Charging Contactor”. Unprotected this can lead as a Direct Consequence to unsymmetrical phase currents, further to the Indirect Consequence of “Conducted EMI in one of the Reed (“Reed” is a particular type of track circuit) bands” and ultimately to the TOP CONSEQUENCE of “Signalling WSF”. The proportion leading to a Credible Indirect Consequence (“Excessive conducted EMI”) is dependent on the Properties Qualifying for Credibility which comprehend information on the critical Reed band frequencies as well as on the limits for the corresponding current levels.
There are three **Internal Barriers**. Two **Internal Protections** in the form of “Contactor Supervision” and “Line Circuit Current Protection” and the mitigating **Internal Condition** of a very limited “Mean Fractional Charging Time”. The ISS/LIM system (see Sec. 5 and 6) provides dedicated **External Protections** against exceedence of limits in the Reed bands. As **External Condition** and additionally mitigating fact we have the limited “Mean Fractional Operating Time on routes with Reed track circuits”.

![Diagram](Fig. 1. Control and Safety Concept)

ii) An “Earth fault in the Line Harmonics Filter” may be another example of **INITIATING EVENTs** (IE). Unprotected this can lead as a **Direct Consequence** to the loss of the filtering capability, further to the **Indirect Consequence** of “Conducted EMI at TI21 (“TI21” is a particular type of track circuit) frequencies” and ultimately to the **TOP CONSEQUENCE** of “Signalling WSF”. The proportion leading to a **Credible Indirect Consequence** (“Excessive conducted EMI”) is depending on the **Properties Qualifying for Credibility** comprehending information on the critical TI21 frequencies as well as on the limits for the corresponding current levels. There are two **Internal Protections** in form of a “Line Harmonics Filter Fuse” and the “Line Harmonics Filter Over-current Protection”. On the other
side, there is no **Internal Condition** to invoke for mitigation. The ISS/LIM system (see Sec. 5 and 6) provides **External Protections** against transgression of limits for TI21 frequencies in form of Broad Band FFT (Fast Fourier Transformation) facilities. As **External Condition** and mitigating factor we have the limited “Mean Fractional Operating Time on routes with TI21 track circuits“.

Depending on the particular situation (project) the rate of occurrence of a **Credible Indirect Consequence**, with or without invoking **External Conditions**, might already be below the limit of **Tolerable Risk**. In this case it is not necessary to provide for any extra **External Protection**. This decision can only be taken in a real situation and this fact underlines the importance of the earliest possible identification of the relevant design criteria as discussed in Sec. 3 above.

### 4.1 New Concepts

We note that there is a conceptual discontinuity in the figure above. Its lower left part can well be described by methodologies belonging to the domain “Functionality” (RAM): an **INITIATING EVENT** (cause/Basic Event) leading to a **Direct Consequence** (Top Event). This is clearly different from the subsequent **Indirect Consequence** whose appearance has no similarities whatsoever with its origin. This fact puts the question whether the same methodologies and available tools are satisfactory to continue the description and allow for a proper analysis of the **Credible Indirect Consequence** or of the **TOP CONSEQUENCE**. The question was dealt with earlier by expanding and complementing the customary set of notions and methodologies [4] (unfortunately, the digital processing of the manuscript made the figures illegible and readable copies can be requested from the author). Traditionally, part of Probabilistic Safety Assessment (PSA) is documented by means of Event Trees and/or Fault Trees (e.g. [5]). However, in practice there are situations where the standard concepts and tools are not sufficient for an adequate characterisation. The latter statement refers specifically to the situation of “**Non-persistent Consequences**”, met when analysing the generation of EMI and the potential consequences.

According to a recent observation [1] both “**Persistent Consequences**” and “**Non-persistent Consequences**” (introduced and discussed in [4]) could be covered by the same model adequate for dealing with “**Mixed Persistency Consequences**”. In that case, due to a PFD (characterised by the failure rate and the length of the “Proof Check Interval”) different from zero, there is a “**Persistent Consequence**” with a rate of occurrence (frequency) corresponding to the relevant failure rate. At the same time (simultaneously), due to the finite “**Reaction Time**” (corresponding exposure time) there is a finite probability for a “**Non-persistent Consequence**” to occur.

The release of fluids, toxic gas or radiation in any form; the initiation of a fire or an explosion are all typical examples of “**Persistent Consequences**”. As still most of the safety issues address these types of hazards, it is not surprising that the need of new and more appropriate concepts is not immediately obvious. However, other types of hazards in connection with modern control and/or protection systems, as discussed above, substantiate the need.
5 Design Strategy

The “EUC risk” related to the generation of EMI by the propulsion system, which directly or indirectly is capable of causing a signalling Wrong Side Failure, was addressed separately from the beginning in the design process. In the line of designing a generic product, the decision was to provide an electrical/electronic/programmable electronic safety-related system (E/E/PES) as an external risk reduction facility in form of a dedicated ISS/LIM (Interference Supervision System/Line Interference Monitor) system with appropriate requirements on the functional safety.

5.1 Control Function: Drive Control

We are aware of the fact that the Drive Control Units as used within the propulsion (converter) assemblies obtain their full significance only when the control (and protection) system is put into its global context. As stated above, this means that the application (project) specific equipment as well as the infrastructure is equally important. Only their comprehensive consideration results in a proper identification of potential “EUC risks” and corresponding requirements for the functional safety. In order to be a generic product that can be used in a wide range of different applications, the controller (DCU) was designed, developed and will be maintained according to the relevant safety standards.

5.2 Safety Function: Line Interference Monitor

The primary task of a Line Interference Monitor, LIM, is to acquire information on potential interference currents, to process the signals and if necessary, initiate protective actions. As an electrical/electronic/programmable electronic safety-related system (E/E/PES), the corresponding design target for the safety integrity level is set to SIL 2 (see [3]). The associated hardware safety integrity given as a Probability of Failure on Demand (PFD) can be assessed in dependence of the length of the relevant proof check interval. As expected, the PFD gets lower if the length of the proof check interval is decreased.

6 Example: Safety Function for the UKE-Project

At the initiation of the UKE-project (Electrical Multiple Units for the UK market, in reversed order) the Drive Control Units were not fully developed. The decision was not to claim less than one failure per ten years corresponding to a failure rate not lower than $1.14 \times 10^{-5}$ failures per hour. More explicitly, the hope was not to have to classify the EUC control system as a safety-related system. Therefore, the “EUC risk” related to the generation of EMI by the propulsion system, which directly or indirectly is capable of causing a signalling Wrong Side Failure, was addressed separately from the beginning in the design process. The decision was to provide an electrical/electronic/programmable electronic safety-related system (E/E/PES) as a
risk reduction facility in form of a dedicated ISS/LIM (Interference Supervision System/Line Interference Monitor) system with appropriate requirements on the functional safety.

As we noted above, the applicable standard [3] allows for this view and possibility as long as the stated requirements are fulfilled. In the UKE-project these requirements are taken into account by:

a) documented (Mean Time Between Failure) MTBF-calculation and follow-up from actual operating experience in similar applications
b) claimed never lower than 1.14 * 10^-5 failures per hour
c) FMEA on the entire propulsion system scope of supply
d) separate and independent signal relays in the trip chain (opening of the Line Circuit Breaker) and back-up by converter blocking

The performed Fault Tree Analyses were based on very conservative “Properties Qualifying for Credibility”. For AC Operation all of the involved nine different types of track circuits resulted in frequencies (rate of occurrence) of the alleged (some of it confirmed by tests) EMI transgressions by one order of magnitude lower than the requirements in the corresponding “Whole Train Risk Model”. This proves that for the chosen design the required safety integrity is achieved more than adequately. Hence, it is not necessary to designate the EUC control system as a safety-related system. Therefore, the Drive Control Units do not have to conform to the standard IEC 61508.

In Sec. 1 and 3 it was stated that the relevant safety requirements have to be translated to measurable engineering quantities. This means, the “accident sequence”: injuries/collision/WSF/conducted EMI/over-current/short circuit (as an example) has to be paralleled with a translation to terms that a designer at the subsystem level is familiar with. Only in this way is it feasible for the design to meet a stated target and even more importantly, a validation will be possible. In the project above, the “Whole Train Risk Model” stopped with a target value for the rate of occurrence of WSF. It then was a formidable task to establish corresponding limits for acceptable EMI currents with respect to the different track circuits, as well as for measuring and analysing the indirect consequences (harmonic currents) of alleged failure modes (IE).

7 Summary and Conclusions

As for the parallel potential contradiction between “Safety” and “Availability”, it was shown that there need not be any conflict between “Control Functions” and “Safety Functions” as long as “Functionality” and “Safety” are integral parts of the design process. The most efficient way is to consider all the issues on an equal basis at the earliest possible stage in the development or design.

With regard to “Safety” analysis we noted that the “Functional Specification” does not usually contain any specific quantitative requirements on “Safety” and that a reason for this is that neither all of the “Safety” issues are known nor are they analysed at that time. As soon as a preliminary technical solution to attain the specified functionality is ready, the related potential hazards have to be identified and
the establishing of a relevant “Risk Model” has to be initiated. This model is needed in order to translate the “Tolerable Risk” based on the current values of society to measurable engineering quantities in form of limits for currents, electromagnetic fields with associated critical frequency ranges and/or exposure times. This is an essential step as these quantities are necessary to initiate the final design. Furthermore, at the beginning the concerned parties even might not be aware of the necessity to acquire and provide the relevant information. Again, the explanation for the latter fact is that quantitative “Safety” requirements are not part of a traditional “Functional Specification”. This summarises one of the main conclusions.

The other major conclusion is that “Control Functions” and “Safety Functions” should be kept separate. Not only for the reason that the applicable standard [3] facilitates this approach, but mainly due to hardware properties. The underlying hardware for “Control Functions” of complex systems (for obvious reasons) tends to contain many components resulting in relative low MTBF values. These cannot be compensated by extremely high requirements on the software Safety Integrity Level (SIL) to be achieved.

A practical example demonstrated the feasibility of this approach. The results showed that adequate safety integrity was achieved and that the target failure measure was far below the value for the corresponding “Tolerable Risk”.

Discussing the transition from the Direct Consequence to the Indirect Consequence we advocated the use of new concepts such as “Non-persistent Consequences” and “Mixed Persistency Consequences”, as well as the ones introduced and discussed in [4]. They are needed for a coherent description of “Safety” issues met in practical (real) applications. The need for new concepts is accompanied by the need for new or adapted tools. This fact is presented as a challenge to the vendors of software packages to provide the analysts with suitable ones.

References

A Fail-Safe Dual Channel Robot Control for Surgery Applications

Ulrich Laible, Thomas Bürger, and Günter Pritschow
Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen, Universität Stuttgart, Seidenstr. 36, 70174 Stuttgart, Germany
ulrich.laible@isw.uni-stuttgart.de

Abstract. This paper presents the architecture of a fail-safe control for robotic surgery that uses two independent processing units to calculate the position values and compare the results before passing them to the drives. The presented system also includes several other safety functions like a redundant measuring system realized as a tripod within the hexapod kinematics, position lag monitoring or watchdogs. The safety requirements for the system are derived from the regulations of the medical device directive (MDD) and from a risk analysis of the control system.

1 Introduction

Numerical Controls for robots are generally safety critical, since they are directly responsible for movements of the robot. More and more robots are used for applications, where they are in direct contact with humans, causing a potential hazard. This is especially true, when the controlled robot is used for surgery applications in an operating room. In this case a safety concept must be applied, which prevents any uncontrolled or unwanted movement of the robot.

Such a safety concept was applied for the control system of a commercial surgical robot system [1] using an approved motion control software library [2]. This paper analyzes the safety requirements for robotic surgery and describes the main safety features of this fail-safe robot control.

2 Application

The controlled robot system (Figure 1) has six driven axes arranged in a parallel kinematics structure (hexapod) and a linear axes mounted on top of the platform. Three passive axes arranged in a tripod structure inside the hexapod are used as a redundant measuring system. The linear axis of the robot system can be used as a universal carrier for various surgical instruments (e.g. endoscop, milling cutter, rasp, drill, drilling jig, etc.). The whole robot system is in turn attached to a carrier-system, which is equipped with further axes in order to pre-position the robot system. These axes are not used during the operation and are controlled separately.
The robot system can be used in an automatic mode and in a manual mode. In the manual mode the robot is controlled with different input devices such as a force torque sensor and a joystick. In the automatic mode, the robot performs movements according to the program file interpreted by the control system.

In contrast to other available surgical robot systems, this system was designed for universal use to assist a wide range of surgical treatments. For neurosurgery or minimal-invasive surgery an endoscop can be moved and positioned in manual mode. To assist during hip replacements the system is equipped with a pneumatic rasper or milling cutter and operates in the automatic mode [5]. Many other applications are also possible.

3 Safety Requirements

3.1 Requirements by Regulations

The surgical robot falls under the scope of the Medical Devices Directive (MDD) [7]. The MDD has to be followed in order to receive the CE marking, which is mandatory for selling a medical device in the European Union. The Competent Authority approves so-called “notified bodies” who are permitted to carry out the conformity assessment tasks.

The MDD divides devices into four classes, Class I - Low risk, Class IIa and IIb - medium risk and Class III - high risk. The devices are classified according to general criteria, particularly invasiveness, duration of continuous contact, nature of the tissue contact and distinction between active and non-active devices. Like all active therapeutic devices that may administer or exchange energy to or from the human
body in a potentially hazardous way, the described robotic surgery system is classified in Class IIb.

The safety requirements are not restricted to patients but also include users and, where applicable, other persons. The MDD also includes other directives [6], which have to be followed. For the numerical control of the surgical robot, the main requirements can be summarized as follows:

- Use of a quality management system
- The performance of a risk analysis
- Meeting EMC and radiation regulations
- Functional safety
- Appropriate performance.

3.2 Medical Requirements

For the intended surgical treatments, the functional safety can primarily be realized by guaranteeing that the system is fail-safe. This means that the control system must be able to detect any failure condition, which could possibly lead to a hazard. If a failure is detected the system must immediately stop all movements (Safe State of the system). For the considered medical applications it has to be ensured that the movement of the tool center point stops within 1 mm. Subsequently the standard procedure is to unlock and remove the instruments and to finish the operation manually.

3.3 Requirements from a Risk Analysis

The implementation of a risk analysis is a substantial part of the safety concept of the complete system and a key requirement of the MDD [8]. A risk analysis identifies hazards and risks associated with the use of the device. During the design phase the risk analysis gives important hints, where special safety functions are required in order to detect a failure and avoid potential hazards. At a later state in the development the risk analysis is also a method by which the safety concept can be verified and the residual risk can be estimated.

For the surgical robot system a FTA [9] and a FMEA [10] were performed. The FTA ended in a branch identifying the control system as a potential cause for hazards. From the other side the complete control system, including hardware, software and drives, was examined in a FMEA (Figure 2). Many requirements for the safety functions detailed in section 4 resulted from the FMEA, one example is shown in the following:

The drives, which are integrated in the hexapod axis, are coupled to an indirect measuring system (see Fig. 2). The encoder is connected to one side of the motor shaft whereas the spindle is coupled on the other side. If the coupling between motor and spindle breaks, the control still receives the encoder signals and does not detect any failure condition. To detect this failure situation a redundant measuring system is required.
4 Safety Technology for Robot Controls – State of the Art

Modern robot controls (RC) for industrial robots and also numerical controls (NC) for machine tools already have a quite high safety standard. Nevertheless most industrial robots are operated behind metal fences and normally no human is allowed to stay in the reach of the robot. Exceptions are setting-up operations or error-fixing, where an operator has to work close to the robot.

Standard safety measures (some required by regulations) of today’s RC/NC are [11]:

– Reduced maximum speed of the robot in setting-up mode
– Confirmation of robot movements using an handheld operator panel
– Fail-safe emergency stop circuit
– Comparing command and actual position values (position lag monitoring)
– Initial testing of RAM and CPU

While safety is becoming more and more important for automated systems, some RC/NC manufactures introduce special safety-variants of their control systems. Most of them are based on the use of a digital drive system [4]. Having a second processor unit in the digital drive system allows the use of watchdog functions, additional plausibility checks and an independent shutdown in case of an error or emergency stop. Also important I/O-signals are read/written redundant and are processed diverse in the control unit and in the servo amplifier. However not the complete control functionality is processed redundant, therefore the systems stay vulnerable to RAM or CPU failures during runtime (i.e. through electromagnetic interference).

In the field of automation technology a multi channel structure with complete redundant processing of the control functionality is today only used for programmable logic controller (PLC) in safety critical areas [3].
5 Fail-Safe Robot Control System

Following the fail-safe-principle two main tasks can be identified for the control system of the surgery robot: Failure detection and Failure Reaction. The failure detection must be able to detect any possible failure condition which could lead to an uncontrolled movement of the robot in a minimum amount of time and subsequently the failure reaction must stop any movements of the robot.

Failure detection and failure reaction can be classified in one-time tests and monitoring functions, which are performed continually. If the monitoring function is realized inside the RC, is here called internal monitoring, otherwise external.

Table 1. Safety functions for the fail safe robot control

<table>
<thead>
<tr>
<th></th>
<th>Failure Detection</th>
<th>Failure Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One-Time Tests</strong></td>
<td>– Initial test</td>
<td>– Error message</td>
</tr>
<tr>
<td></td>
<td>– Lock all output signals to actors</td>
<td></td>
</tr>
<tr>
<td><strong>Internal Monitoring</strong></td>
<td>– Dynamic position lag monitoring</td>
<td>– Feed-hold</td>
</tr>
<tr>
<td></td>
<td>– Redundant measuring system</td>
<td>– Error message</td>
</tr>
<tr>
<td></td>
<td>– Feedrate monitoring</td>
<td>– Power shutdown by control channel</td>
</tr>
<tr>
<td></td>
<td>– Cycle-time monitoring</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– Software limit switch monitoring</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– Dynamic range monitoring</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– Plausibility checks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– Exception handler</td>
<td></td>
</tr>
<tr>
<td><strong>External Monitoring</strong></td>
<td>– Monitoring channel</td>
<td>– Emergency stop by operator</td>
</tr>
<tr>
<td></td>
<td>– Uninterrupted power supply (UPS)</td>
<td>– Power shutdown by monitoring channel</td>
</tr>
</tbody>
</table>

5.1 Initial Test

During the system start-up many system components are tested automatically or interactively. This includes:

- Memory and CPU test by the extended BIOS
- Checksum test of the executable code and the parameter files
- Interactive test of the peripheries (joystick, force sensor, confirmation button)
- Test of the drive system
- Test of shutdown paths (feedhold, power shutdown, emergency stop button)

5.2 Dynamic Position Lag Monitoring

The Position Lag Monitoring (PLM) monitors the difference between position value commanded by the control and the actual position value measured by the measuring
PLM is essential in order to detect failures in the drive system, such as for example
- Interruption of the connections to the drives or from the measuring system,
- Failure in the servo amplifier,
- Failure in I/O modules,
- Power loss of encoder.

To minimize the reaction time of the failure detection, this control system uses a dynamic PLM. The limit for the allowed position lag is changed dynamically according to the current velocity and acceleration. If the axis is stopped, the limit for the allowed position lag is almost zero and the PLM has the functionality of a standstill monitoring.

### 5.3 Redundant Measuring System

One element of the safety concept is the redundant measuring system, which allows the monitoring of the platform position with two independent measuring systems. Since the second measuring system is part of the tripod kinematics and the primary measuring system is integrated into the hexapod, the kinematical transformation can be calculated in two different ways and is therefore redundant and diverse. The second measuring system also uses different hardware (counter boards), which enables the system to detect any failure in the primary measuring system (Figure 4). Also the linear axis of the robot system, that is controlled using an indirect measuring system, is monitored through an additional direct measuring system.

Using a different kinematics for the redundant measuring system allows also the detection of additional failure conditions: Setting the home position of the robot to a wrong position value during the homing procedure on system startup would result in a wrong absolute position of the robot in following robot movements. This failure can be detected by the redundant measuring system, since the kinematical transformations for both kinematics produces different results when using the wrong home position as reference. The failure detection is dependent on the used tolerance, the performed movement and the offset of the real home position. To detect this failure reliably the robot must perform movements using the whole workspace after the homing procedure.

### 5.4 Dual Channel Architecture

All safety functions mentioned so far are realized inside the control unit. There are however failure conditions which cannot be detected using a single channel system with one CPU [12]. These conditions are wrong processing of position values through
- Coincidental CPU failure during runtime,
- Failures in RAM during runtime,
- Electromagnetic interference
To detect these failures a second NC channel with redundant hard- and software is being used (Figure 4). The second channel has the same full functionality as the first one, but is based on a different hardware, minimizing the risk of undiscovered production failure.

A second independent implementation of the complete software for the second channel is economically not feasible. Therefore software errors have to be minimized by using an approved RC software that is integrated in a quality assurance system (section 6).

The second channel (monitoring channel) performs exactly the same calculations as the first one (control channel), so that the computed command position values of both channels can be compared with each other. Since the monitoring channel always runs three cycles before, the control channel only transfers position values to the drives, which have been checked by the monitoring channel.

The operator control computer commands both channels independently with the same data. This data is also compared and only passed if equal. Since these commands are not processed in real-time, synchronization mechanisms have to be provided in the non-real-time part of the NC control software. On the other hand in the real-time part the monitoring channel never must stop the control channel.
5.5 Failure Reaction

If a failure of the robot system has been detected by any of the internal or external monitoring functions, a failure reaction is initiated. This failure reaction can be performed independently by the control channel and by the monitoring channel.

Depending on the nature of the detected failure different reactions are initiated. If the cause of the error does not affect the ability to control the drives properly by the control system, the axis are stopped by the control keeping the desired path. If the detected error is serious and could possibly result in a uncontrolled movement the reaction manager shuts down the power supply of the drive via the emergency stop circuit. Through self-locking the axes stop in a tolerable time.

Considering the medical requirements (chapter 3.2) the axis have to be stopped within a maximum deviation of 1 mm measured at the tool center point. The RC runs with a cycle time of 4 ms, so the reaction time after the detection of a failure is less than 4 ms. The power shutdown for the drives takes additional 8-12 ms. Failure simulations have shown, that in a worst case condition with maximum speed the deviation at the tool center point (TCP) is less than 1 mm, measured from the point of time the failure is detected.
6 Quality Assurance

The RC software was build up on commercial standard control software for motion control, which is currently in use in many different applications for machine tools and industrial robots. Since the control software is in constant use since several years it is classified as approved by the competent authority. The same is true for the used operating system VxWorks, which is already in use in many safety critical applications.

Additionally all used software must be integrated in an adequate quality assurance (QA) system, which is also one requirement to receive the CE marking for the control system. Beside the constructive QA activities (configuration management, guidelines, etc.), the main focus during the implementation of the RC was put on the analytical QA activities (test, review, simulation, etc.). Automatic functional tests were executed in order to detect defects in the software during the development stage. Furthermore failure simulation tests were carried out in order to test the fail-safe behavior of the surgery robot system.

6.1 Automated Regression Testing

As the development of the control software for the surgery robot is an extension and adaptation of existing control software components, regression testing is an appropriate method for verifying the basic functionality of the control software [13]. The reliability of the control software components, which were used as basis for the necessary extensions, can be proved by evaluating the number and effect of known software faults in industrial used versions of the software.

The desired behavior of the extended control software for a regression test can be defined by the execution of the same test cases with the already approved basis control software (reference).

If a subsequent variance comparison of the actual test results of the current development version and the approved version shows differences which are not related to the intended changes in the software, they are an indication of defects within the current version of the control software.

To carry out this kind of functional test in an economical way, it is important to execute these tests automatically and reproducible. Therefore the complete test execution (processing of test cases, acquisition of test results, variance comparison) is automated. To obtain a reproducible test sequence it is important to generate comparable test results. This requirement can be complied using a modular structure for the control software. Due to the encapsulation of functionality within functional units, the access to defined data interfaces is possible. Using these interfaces the internal process data flow and control data flow can be recorded and used as test results for variance comparison.

6.2 Verification of the Fail-Safe Behavior

The verification of the fail-safe behavior of the surgery robot system is an essential part of the analytical QA activities and is done by the execution of failure simulations.
For this purpose safety critical defects are intentionally introduced to hardware or software components e.g. by manipulating wires or the RAM. The kind and number of defects, used for failure simulations, are derived from the FMEA. For each failure the corresponding defect or defects have to be initiated in order to verify the detectability and the appropriate corrective action.

7 Summary

All safety functions described within this paper were realized in a commercial control system as part of the robotic surgery system of the company Universal Robot Systems GmbH (URS). The control system was realized on a cPCI-System with two CPU boards and five I/O boards.

This system was certified by a German notified body (TÜV Product Service) according to the corresponding regulations and is currently in the final process of receiving the CE marking.

The development has shown that a dual channel motion control system is technical and economical feasible, if very high safety standards are required. The control system cannot only be used for surgery robots but also for any kind of robot system or machine tool with controlled servo drives.

Fig. 5. Robotic Surgery System (Photo: URS)
References

10 IEC 60812, 1985, Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA).
Abstract. Human operators use mental models to guide their interaction with automated systems. We can “model the human” by constructing explicit descriptions of plausible mental models. Using mechanized formal methods, we can then calculate divergences between the actual system behavior and that suggested by the mental model. These divergences indicate possible automation surprises and other human factors problems and suggest places where the design should be improved.

1 Introduction

Human error is implicated in many accidents and incidents involving computerized systems, with problems and design flaws in the human-computer interface often cited as a contributory factor. These issues are particularly well-documented in the cockpits of advanced commercial aircraft, where several fatal crashes and other incidents are attributed to problems in the “flightcrew-automation interface” [8, Appendix D].

There is much work, and voluminous literature, on topics related to these issues, including mode confusions [22] and other “automation surprises” [23], human error [18], human cognition [16], and human-centered design [1].

The human-centered approach to automation design explicitly recognizes the interaction between human and computer in complex systems, and the need for each side of the interaction to have a model of the other’s current state and possible future behavior.

“To command effectively, the human operator must be involved and informed. Automated systems need to be predictable and capable of being monitored by human operators. Each element of the system must have knowledge of the other’s intent” [11, Chapter 3].

Computer scientists might recognize in this description something akin to the interaction of concurrent processes, and might then speculate that the combined behavior of human and computer could be analyzed and understood in ways that are similar to those used to reason about interacting processes. In the “assume-guarantee” approach [13], for...
example, each process records what it assumes about the other and specifies, in return, what it will guarantee if those assumptions are met. Now, the computer side of this interaction is, or can naturally be modeled as, a process in some formal system for reasoning about computational artifacts. But what about the human side: is it reasonable to model the human as a computational system?

It turns out that modern cognitive science holds the view that the mind is, precisely, a computational system (or, at least, an information processor) of some kind. Thus, we can imagine constructing a computational model of some aspects of human cognition and behavior, confronting this with a similar model of the computerized system with which it is to interact, and using formal calculation to derive observations or conclusions about their joint behavior.

Explicit models of human performance have long been used in computer interface design: for example, GOMS (Goals, Operators, Methods, and Selections) analysis dates back to 1983 and has spawned many variants that are used today. More recently, cognitive models have been used to simulate human capabilities in systems for developing and evaluating user interfaces. Deeper models such as ICS (Interacting Cognitive Subsystems) allow examination of the cognitive resources required to operate a particular interface. These approaches are useful in identifying error-prone features in interfaces to safety-critical systems (e.g., the complex process that must be followed to enter a new flight plan into a flight management system), but they do not seem to address the most worrying kinds of problems: those associated with mode confusions and other kinds of automation surprise.

Automation surprises occur when an automated system does not behave as its operator expects. Modern cognitive psychology has established the importance of mental models in guiding humans interaction with the world; in particular, operators and users of automated systems develop such models of their system’s behavior and use these to guide their interaction. Seen from this perspective, an automation surprise occurs when the actual behavior of a system departs from that predicted by its operator’s mental model.

Mental models of physical systems are three-dimensional kinematic structures that correspond to the structure of what they represent. They are akin to architects’ models of buildings and to chemists’ models of complex molecules. For logical systems, it is uncertain whether a mental model is a state transition system, or a more goal-oriented representation (e.g., chains of actions for satisfying specific goals). There is some experimental support for the latter view, but this may depend on how well the operator understands the real system (with deeper understanding corresponding to a more state-centered view). In any case, a mental model is an approximate representation of the real system—an analogy or imitation—that permits trial and evaluation of alternatives, and prediction of outcomes. Being approximate, it is bound to break down occasionally by “showing properties not found in the process it imitates, or by not possessing properties of the process it imitates.” In principle, we could attempt (by observations, questionnaires, or experiments) to discover the mental model of a particular computerized system held by a particular operator, and could then examine how it differs from the real system and thereby predict where automation surprises might occur for that combination of system and operator.
It is, of course, difficult and expensive to extract the individual mental models needed to perform this comparison. Fortunately, it is not necessary (although it might be interesting for experiment and demonstration): most automation surprises reported in the literature are not the result of an errant operator holding a specific and inaccurate mental model but are instead due to the design of the automation being so poor that no plausible mental model can represent it accurately. Quite generic mental models are adequate for the purpose of detecting such flawed designs. In the next section, I propose methods for constructing such mental models and for using them to guide development of systems that are less likely to provoke automation surprises.

2 Proposed Approach

Generic mental models can be constructed as state machines whose states and inputs are derived from information available to the operator (e.g., the position of certain switches and dials, the illumination of certain lamps, or the contents of certain displays), information in the operators’ manual, and the expectation that there should be some reasonably simple and regular structure to the transitions. If a mental model is an accurate representation of the real system, there should be a simulation relationship between its state machine and that which describes the real system. Proposed simulation relations can be checked automatically using model checking or reachability analysis: these explore all possible behaviors by a brute force search and will report scenarios that cause the simulation relation to fail.

Colleagues and I have used this kind of analysis to explore automation surprises in the autopilots of the MD-88 [20], A320 [5], and 737 [21]. In each case, a plausible mental model exposed exactly the scenarios that have led to reported surprises and consequent “altitude busts,” and pinpointed elements in the behavior of the actual system that preclude construction of an accurate mental model (because the behavior of the actual system depends on state transitions that are invisible at the user interface).

These experiments have convinced me of the basic efficacy of the approach, but the exciting opportunity is to move beyond detection of known flaws in existing systems to the development of a method that can be used to predict and eliminate such flaws during design. For this purpose, we need a systematic and repeatable method for constructing generic—yet credible—mental models. Work by Javaux suggests the general “shape” of such models and a process to create that shape [9].

Javaux proposes that training initially equips operators with fairly detailed and precise mental models. Experience then simplifies these initial models through two processes. The process of frequential simplification causes rarely taken transitions, or rarely encountered guards on transitions, to be forgotten. The process of inferential simplification causes transition rules that are “similar” to one another to be merged into a single prototypical rule that blurs their differences. We can imagine a computer program that applies these simplifications to turn the representation of an initial mental model into one for a more realistic “mature” one.

Given such a program that mechanizes Javaux’ simplifications, I propose the following approach to development of automated systems.
– Construct a representation (i.e., a formal model, a simulation, or running code) of the actual automation design.
– Construct an initial mental model.

This could be based on the instruction manual for the proposed design, or constructed by a process similar to that used to develop an instruction manual, or it could even be taken directly from the actual system design.
– Check the initial mental model against the actual design.

Using model checking techniques similar to those described previously [20,5,21], check whether the initial mental model is an adequate description of the actual system. If so, proceed to the next step, otherwise modify the design and its initial mental model and iterate this and the previous steps (there is no point in proceeding until we have some description that accurately reflects the actual system).
– Construct a simplified mental model.

Use a mechanization of Javaux’s two processes to simplify the initial mental model into a more realistic one.
– Check the simplified mental model against the actual design.

Using model checking techniques, check whether the simplified mental model is an adequate description of the actual system. Terminate if it is, otherwise modify the design and iterate this and the previous steps.

The outcome of this process should be a system design whose visible behavior is sufficiently simple and regular that an operator, guided only by externally visible information, can accurately predict its behavior and thereby interact with it in an informed and safe manner. Furthermore, the simplified mental model produced in the process can provide the basis for an accurate and effective training manual.

It is important to note that the point of this process is not to construct a mental model that is claimed to be faithful to that of any particular operator, but to use what is known about the characteristics of mental models to coerce the design of the actual automation into a form that is capable of supporting an accurate mental model.

3 Conclusion

To predict the joint behavior of two interacting systems, we can construct formal models for each of them and calculate properties of their combination. If one of the systems concerned is a human, then we can extend this approach by modeling computational aspects of human cognitive functions. For the case of human operators of automated systems, it is known that they use simplified representations of the system as a mental model to guide their interaction with it.

The mode confusions and other automation surprises that are a source of concern in operator’s interactions with many automated systems can be attributed to appallingly bad designs that admit no plausibly simple, yet accurate, mental models. By “modeling the human”—that is by explicitly constructing generic mental models, and by mechanizing plausible processes that simplify them in ways characteristic of real mental models—we can construct a touchstone that highlights cognitively complex aspects of proposed designs and guides their reformulation in ways that promotes simplicity and regularity.
and hence—it is hoped—reduces the number and severity of human factors problems that they provoke.

This approach suggests a number of interesting possibilities for modeling and analysis in addition to those already illustrated.

– We can examine the consequences of a faulty operator: simply endow the mental model with selected faulty behaviors and observe their consequences. The effectiveness of remedies such as lockins and lockouts, or improved displays, can be evaluated similarly.

– We can examine the cognitive load placed on an operator: if the simplest mental model that can adequately track the actual system requires many states, or a moderately complicated data structure such as a stack, then we may consider the system too complex for reliable human operation. We can use the same method to evaluate any improvement achieved by additional or modified output displays, or by redesigning the system behavior. This could provide a formal way to evaluate the methods proposed by Vakil and Hansman for mitigating the complexity of interfaces [24].

– We could take a mental model from one system (e.g., an A320) and check it against a different actual system (e.g., an A340). Discrepancies could highlight areas that should be given special attention in training programs to convert operators from one system to the other.

– We could extend the approach to multi-operator systems: for example, the air traffic control system, where the controller and the pilot may act according to different mental models of the same situation.

References


Papers on formal methods and automated verification by SRI authors can generally be located by a search at http://www.csl.sri.com.

Oliver Mäckel and Georg Hoever

Siemens AG
Corporate Technology, CT PP 2
Simulation and Risk Management
D-81730 Munich, Germany
{Oliver.Maeckel, Georg.Hoever}@mchp.siemens.de

Abstract. Safety-critical systems are influencing human life everywhere. During the development of these safety-critical systems the detection, analysis and avoidance of technical risks is essential. While most methods only consider possible failures and faults of a system itself, here a systemic and basic applicable technique is presented, to find out critical and problematic human-machine interactions during the operation of a system. The technique is based on a state-interaction matrix, carefully filled out and afterwards automatically evaluated.

1 Motivation

Nowadays technical systems are wide-spread in the life of human beings. Always more and increasingly more complex tasks are taken over by machines. Humans still control the processes. In particular these human-machine interactions (HMI) are very important in the environment of safety-critical systems. Accidents due to incorrect human-machine interactions often result in physical damage, injured or even dead persons [2].

For example, in the USA and Canada some accidents occurred during the operation of the Therac-25 - a medical radiation device for destroying tumors - in the years 1985-1987: Patients received a huge radiation overdose. The reason were some misleading possibilities for the setting and resetting of the amount of the radiation. These possibilities of HMI had not been recognized during the development of the system [2, 3].

A further example of an imperfect analysis of HMI is the bombardment of an Iranian airbus through the USS Vincennes on the 3rd July 1988 at which all 290 passengers were killed. Cause was an imperfect designed man-machine interface [2].

Another example is the frontal crash of two trains near Berlin on Good Friday 1993, resulting in three dead and more than 20 injured persons. During the days before the route had been used in single-line operation because of construction works. On Good Friday the normal double-line operation should be used. The area manager

1 A system [1] is a combination of technical and organizational measures and things for the autonomous fulfillment of a function.
(in German: Fahrdienstleiter) failed to adjust the train control system correctly. This did not cause the accident, because the automatic control system sets the signal on stop for the first train. But the area manager thought that this reaction was a system failure and used an additional signal, planned for special situations in the case of construction works in order to allow the continuation of the operation. He overlooked that an additional, not scheduled train approached from the opposite direction [4].

These examples show that a majority of problems concerning complex technical systems have their origins during the early phases of the system development – the requirements specification phase - and related to the different human-machine interaction possibilities during the system operation. More and more often these possibilities are examined imperfectly in the requirements specification phase.

To avoid these risks the HMIs should be precisely analyzed during the requirements specification phase. Indeed, such analyses are required by relevant standards [5, 6]. However, they are established only rudimentarily in practice. In the following we present a systemic applicable technique to analyze the possible HMIs of a system.

2 Introduction

2.1 Examples

Normally, there are a lot of possibilities for HMIs. E.g., a door can be opened or closed, as well as locked or unlocked. For the normal use the order of HMIs is important. So it is reasonable, first to close an open unlocked door and then to lock it. The reverse sequence - first locking the door, and then closing it, leads to an other, mostly not intended situation: With the key-bolt sticking out, the door can not be closed completely and there could arise any damage on the door-frame due to trying to close the door completely.

Of course, also the actual state of a system is important for the consequences of an interaction. For example, it is useful to loosen the stirring spoon of a mixer (e.g., to clean them), but only when the mixer is switched off. During stirring, the same action can be very dangerous.

As shown it depends on the current state of the system and the specific order, which HMIs are reasonable and which are not. The technique shown in the following describes how problematic and critical (sequences of) HMIs can be recognized systematically and also in which kind a human-machine interaction analysis is able to give reasonable input for a system requirements specification.

2.2 Human-Machine Interactions Analysis

A human-machine interaction analysis should examine, whether an interaction or a sequence of interactions can bring the system into a critical or problematic state. It is supposed to show critical HMIs in a systemic and complete way. These can afterwards be handled and, e.g., prevented by technical or organizational measures.
2.3 Difficulties

As the door example showed, in principle you have to take into account all sequences of interactions. This corresponds to the process of an Event Tree Analysis [6]. Further, the effects of the different human-machine interaction sequences depends on the respective initial state (see the mixer example above). But in fact you need not to carry out a complete Event Tree Analysis for every system state. We want to explain this in the following.

3 Safety Oriented HMI Analysis

First of all, you have to answer the following two questions:
- Which states does the considered system have?
- Which HMIs have to be considered?

3.1 States

In order to get systemically all states of a system, the system has to be decomposed into single components. Then, individual states to these components are assigned. Now, each combination of these states (for each component one state) results in a state of the whole system. These combinations can be generated systemically / algorithmically. Normally, you can declare many of these combinations to be impossible because of technically or physically reasons. Other combinations could be physically possible but forbidden or not intended in practice. For example, let us consider the brake and gas pedal as well as the parking brake of a car. The pedals possess the individual states "pressed" and "not pressed", the parking brake the single states "pulled" and "not pulled". Hence eight combinations result (see Table 1).

<table>
<thead>
<tr>
<th>State</th>
<th>brake pedal</th>
<th>gas pedal</th>
<th>parking brake</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pressed</td>
<td>pressed</td>
<td>pulled</td>
</tr>
<tr>
<td>2</td>
<td>pressed</td>
<td>pressed</td>
<td>not pulled</td>
</tr>
<tr>
<td>3</td>
<td>pressed</td>
<td>not pressed</td>
<td>pulled</td>
</tr>
<tr>
<td>4</td>
<td>pressed</td>
<td>not pressed</td>
<td>not pulled</td>
</tr>
<tr>
<td>5</td>
<td>not pressed</td>
<td>pressed</td>
<td>pulled</td>
</tr>
<tr>
<td>6</td>
<td>not pressed</td>
<td>pressed</td>
<td>not pulled</td>
</tr>
<tr>
<td>7</td>
<td>not pressed</td>
<td>not pressed</td>
<td>pulled</td>
</tr>
<tr>
<td>8</td>
<td>not pressed</td>
<td>not pressed</td>
<td>not pulled</td>
</tr>
</tbody>
</table>

The states 1 and 2 are not possible as long as the driver does not use also the left foot, since the right foot can only operate at the brake pedal or at the gas pedal. A pressed gas pedal with pulled parking brake, state 1 and state 5, is technically possible but
mostly not intended. This combination could be prevented automatically. But since in some cases a pressed gas pedal with pulled parking brake is useful (e.g., when starting at a hill) such a prevention is not realized. Instead of this, the most car models have a warning lamp to draw the drivers attention to the possible problems.

3.2 Human-Machine Interactions

After setting up a complete list of the allowed and impossible (or forbidden) states of a system as described above, instead of considering all possible sequences of HMIs, it is indeed enough to analyze a single interaction starting at each possible system state. Each such HMI should reach again an allowed state of the system. If there is a critical sequence of HMIs, there is always a first action that leads from an allowed state to a forbidden one. Analyzing every combination of allowed states and possible HMIs, you will find out also this critical combination, that leads out of the set of the allowed states (see Figure 1). Thereby the analysis of sequences of HMIs is unnecessary.

3.3 States-HMI-Matrix

The analysis of the combinations between the states and HMIs can be done most conveniently in matrix-form. We want to show the procedure by the example mentioned in the introduction: We consider a door, that can be opened and closed and might be locked and unlocked. We will see, how the critical HMIs (closing of a door with the key-bolt sticking out) is revealed automatically.

(Of course, realistic systems are more complex, see e.g. the mentioned example below for the arising size. We are using this simplified example to point out and stress the main points of the approach.)

We consider the single components “door” with the states “opened” and “closed” and also the component “key-bolt”, whose position is determined by the states “unlocked” or “locked”. The possible actions are “unlocking” and “locking”, as well

![Fig. 1. States and interactions](image-url)
as “opening” and “closing”. Hence, the States-HMI-Matrix as shown in Table 2 results. This matrix is now filled systemically (see Table 3): All states are possible and allowed. However opening of an opened door is not reasonable (1C, 2C), according to closing a closed door (3D, 4D), similarly when considering (un-)locking (1A, 3A, 2B, 4B). Action A leads from State 2 to State 1 and from State 4 to State 3. Vice versa, by Action B the first state is transferred into the second one and the third state into the fourth one. The Action C starting at State 3 results in State 1. In the fourth State this action is not possible. This usually corresponds to the required functionality of a door with door lock: When the door is closed and locked it cannot be opened.

Table 2. States-HMI-Matrix

<table>
<thead>
<tr>
<th>Single state of Human-Machine Interaction</th>
<th>State door key-bolt</th>
<th>A: unlocking</th>
<th>B: locking</th>
<th>C: opening</th>
<th>D: closing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 opened unlocked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 opened locked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 closed unlocked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 closed locked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Action D transfers the first to the third state. Finally when analyzing State 2 and Action D we reach the above mentioned critical action: The outstanding key-bolt can damage the door frame when trying to close the door. To handle such a case, you can define different measures:

- Making the corresponding action technically impossible,
- warning the user explicitly or
- declaring the corresponding state as critical.

Table 3. Filled States-HMI-Matrix including a not intended HMI

<table>
<thead>
<tr>
<th>Single state of Human-Machine Interaction</th>
<th>State door key-bolt</th>
<th>A: unlocking</th>
<th>B: locking</th>
<th>C: opening</th>
<th>D: closing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 opened unlocked not reasonable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 opened locked not reasonable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 closed unlocked not reasonable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 closed locked not possible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We want to pursue the last alternative here and define the state of an opened locked door as critical and hence as forbidden (see Table 4). Now, Action B transfers the first state to a forbidden one (this recognition can be done automatically, cf. the following section). But this action (1B) is not immediately combined with dangerous risks: A real damage can arise only after doing a second action (the closing). Further, an attentive user can manifest the critical State 2. Hence, a corresponding prohibition can be sufficient in this case so that no technical prevention has to be planned. In the automotive industry, for example, two independent activities for the locking of the steering wheel lock are considered as sufficient to prevent a possible misuse [7].
### Table 4. Filled States-HMI-Matrix including a critical state

<table>
<thead>
<tr>
<th>State</th>
<th>door</th>
<th>key-bolt</th>
<th>Human-Machine Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>opened</td>
<td>unlocked</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>opened</td>
<td>locked</td>
<td>critical state</td>
</tr>
<tr>
<td>3</td>
<td>closed</td>
<td>unlocked</td>
<td>not reasonable</td>
</tr>
<tr>
<td>4</td>
<td>closed</td>
<td>locked</td>
<td>3</td>
</tr>
</tbody>
</table>

3.4 Automated Evaluation

As mentioned, real systems are normally more complex than the above example. E.g., for a practical analysis developed by Siemens, a subsystem of a public transport system was divided into six components, that possess up to six individual states, which led to 288 total states. A numbering of these states is not reasonable since we would have to search for each number of the resulting states in a tiresome way when filling the matrix. Hence, it is useful to introduce codes for the single states (in the above example for instance “o” and “c” for an open and closed door, according to “u” and “l” for the door lock states) and to sign the states using these codes (the first state corresponds to “ou”, the second one to “ol” and so on).

### Table 5. Filled States-HMI-Matrix, using codes and a substituting symbol “x”

<table>
<thead>
<tr>
<th>Single state of Human-Machine Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>ou</td>
</tr>
<tr>
<td>ol</td>
</tr>
<tr>
<td>cu</td>
</tr>
<tr>
<td>cl</td>
</tr>
</tbody>
</table>

During the above mentioned realistic investigation, 25 relevant HMIs are identified. Considering a certain interaction, in the most cases only one single component changed its state. Hence, it is reasonable, to introduce a substitution symbol, for example “x”, that says that the appropriate individual component state does not change. This simplifies filling out the matrix, because now, we have only to mention the changing code symbol explicitly (see Table 5).

A simple algorithm can substitute the entered “x” by the actual states of the components, e.g. “xl” in the field 1B by “ol”. Now, these codes can be automatically compared with the codes of the forbidden states, so that critical combinations of states and actions reveal automatically (see Table 6).

3.5 Approach

It is suitable, to complete the states-HMI-matrix in a team, similarly to a FMEA, because on the one hand specialized knowledge of several experts is needed for more
complex systems, in order to assess the effects of the HMI. On the other hand the discussion during the analysis supplies - as the experience shows - possible impacts due to the different views of the system. In this way the states-HMI-matrix is validated with respect to the different views.

Table 6. Filled States-HMI-Matrix with replaced codes, including a detected action leading to a critical state

<table>
<thead>
<tr>
<th>State</th>
<th>Single state of</th>
<th>Human-Machine Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ou</td>
<td>opened</td>
<td>unlocked</td>
</tr>
<tr>
<td>ol</td>
<td>opened</td>
<td>locked</td>
</tr>
<tr>
<td>cu</td>
<td>closed</td>
<td>unlocked</td>
</tr>
<tr>
<td>cl</td>
<td>closed</td>
<td>locked</td>
</tr>
</tbody>
</table>

The human-machine interaction analysis should be carried out in following steps:
1. The system that shall be examined is divided into its components, in order to identify the system states as described in Section 3.1. Forbidden, impossible and critical states are marked.
2. The relevant HMI of the system are listed and the state-HMI-matrix is set up.
3. Now, the states-HMI-matrix for all permitted states is filled.
4. If, after an evaluation as described in Section 3.4, critical human-machine interactions are identified, measures should be specified for the improvement of the system by avoidance of these actions, e.g. by technical or organizational measures. Possibly this leads to the declaration of some more critical states (as in the example above), so that the evaluation should be repeated.

4 Conclusion

A HMI analysis as described above is systemic, complete and feasible in practice. The results can be used to specify requirements to the system. E.g., beside of the normal functionality of a door (including the blockade of a closed locked door) it is documented that physical or organizational measures have to be considered to avoid unintended, critical or even dangerous actions (see Table 6).

In safety-critical analyses, it becomes increasingly more important to consider also “human failures” and their effects next to the failures of hardware and software. This has to be taken into account when developing technical systems. The presented method offers a suitable way to do that.

References

7. European Community Directive 95/96/EG
Analysis of Incidents Involving Interactive Systems

Alberto Pasquini\textsuperscript{1}, Antonio Rizzo\textsuperscript{2}, and Luca Save\textsuperscript{2}

\textsuperscript{1} ENEA, Via Anguillarese 301, 00060 Roma, Italy
pasquini@casaccia.enea.it
\textsuperscript{2} University of Siena, Via dei Termini 6, 53100 Siena, Italy
rizzo@unisi.it
save@media.unisi.it

Abstract. This paper describes a methodology for the analysis of the passage of red signals by train drivers (SPAD). The methodology considers all the components that support the train driver in performing his activity and analyses the role of the interactive systems, such as computer based equipments, to understand possible human-machine interaction problems. The application of the methodology has shown that a strong involvement of the users of the interactive systems is a key element to identify their real operative usage, and that this is sometimes quite different from the one hypothesized by the system designers. The paper reports the results of the analysis of an incident to illustrate the application of the methodology.

1 Introduction

For many years the investigation of incidents and accidents concerning safety critical systems identified the “human element” as the major contributing factor. Statistics from various industries attribute about 70\% of accidents to human error [1]. Similar figures hold in transportation, for example in aviation [2] or railway [3]. However, accidents are rarely due to a single cause like a single error by an operator, pilot or driver. In the majority of cases an accident is the result of a complex sequence of events. According to the organisational model of accidents proposed by Reason [4], active and latent failures are both main contributors to the accident. The latter are the results of events that happened in the past, and that created conditions that have not yet been discovered or that are not yet completely realized. Latent failures are decisions or actions, the damaging consequences of which may lie dormant for a long time (i.e. wrong design decisions concerning a human machine interface, wrong management decisions, ineffective procedures), only becoming evident when they combine with local triggering factors (such as technical faults, and atypical system conditions). They are opposed to the more immediately visible failures, which are essentially due to technical problems and called active failures. In particular, active failures are defined by Reason as errors and violations having an immediately and adverse effect upon the system’s integrity.

Some methodologies for the investigation of accidents, proposed in recent years, try to focus not only on the immediately visible failures, but also give adequate attention to the latent failures [5]. The methodology proposed in this paper follows this approach, trying to catch latent and active failures by analyzing all the different
resources that contribute to the functions of the system under investigation: hardware and software; operators; guidelines, procedures and rules used to support and regulate the activities of the operators, documentation, and the knowledge provided by training. Particular emphasis is on the interaction between resources since several authors, see for example [6], [7], and [8] for the specific application field of this study, have shown that most of the failures manifest themselves at the interactions between resources. This concerns both failures that have been classified as active and those classified as latent, according to the Reason’s model.

2 Methodology for SPAD Investigation

SPADs are relatively frequent events. The report of the UK Railway Inspectorate concerning 1998 [9] describes, for that year, over 630 signals passed at danger for a variety of reasons including drivers failing to observe the red signal and the previous warning yellow signal. The vast majority of SPADs involve the train passing the signal by just a few metres, with no danger for the train or its occupants, but in some cases consequences were extremely severe [3]. The methodology presented on this paper has the main objective of promoting reactive actions that can eliminate latent failures, reducing the probability of future SPADs. It has been developed for the analysis of the SPADs that did not have consequences in terms of physical damages or injuries to humans. The absence of consequences paves the way for an analysis that is not influenced by the penal and psychological factors that would be present in case of more severe events. The analysis can be more objective and directed towards the identification of the causes of the incidents, rather than towards the identification of the responsible. The phases of the methodology are briefly described in the following.

Modelling of Process and Resources Identification. The aim of this phase is to identify all the resources involved in the sequence of events leading to the SPAD, and the respective roles of these resources. Resources usually include rules, forms, guidelines, booklets, safety equipment, automated systems and, obviously, the people involved such as train drivers, station manager, level crossing operators. The identification of the resources and of their roles is obtained from the railways experts while modelling the main processes and sub-processes involved in the events. A model shows the sequence of actions and the exchange of information and commands between the resources. A simplified example of model for the process "signal crossing" is shown in fig. 1.

The main resources involved are the train driver, the signal, the train braking system and a computerised system called (on board) Signal Repetition System. Several additional resources, listed in the last column of fig. 1, support the process and can be used, if needed, by the train driver. A detailed explanation of the model presented in fig. 1, and of its notations is provided in the next Section. The modelling of the process is rather time consuming, but the basic processes involved in different SPADs are often the same, thus once a model has been defined, it can be re-used for the analysis of several other types of SPAD, with the needed adjustments and specification. Tables that list the typical resources, and checklists are used to support the modelling and the identification of all the resources involved.
### PROCESS DESCRIPTION FORM - Process: Signal Crossing

<table>
<thead>
<tr>
<th>Sub processes</th>
<th>Main resources involved in the process</th>
<th>Knowledge owned by the driver</th>
<th>Other resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train driver</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal repetition system</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Braking system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver warning</td>
<td></td>
<td>Driver pre-alerted of signal status</td>
<td>-Circulation manual</td>
</tr>
<tr>
<td>Signal acknowledge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal Recognition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal Recognition</td>
<td></td>
<td>Driver aware of signal status</td>
<td>-Train form -Circulation manual -Line booklet -Train course form</td>
</tr>
</tbody>
</table>

**Fig. 1.** Simplified model of the process: Signal Crossing

**Analysis and Comparison of Theoretical and Real Resources Interactions.** The real operating procedures of the process under investigation must be different from the theoretical ones for a SPAD to happen. This may be due to several causes, or as discussed in the Introduction, to a combination of causes, and requires that resources do not interact as planned in process design. The aim of this phase is to compare the theoretical and real interactions of the resources. Theoretical resource interactions can be derived using the model, the list of resources and their characterisation, defined in the previous phase. The real operative resource interactions, for the case under investigation, can be derived from the objective documentation and recording concerning the events leading to the SPAD. This includes for example: the map of the rail line of the zone, the train tachograph recording, the train course form, the interviews with the involved people. This information is used to prepare a graphic representation of the SPAD conditions. The graphic representation includes both the objective, general conditions such as the characteristics of the rail track, and those that are specific of the case under investigation such as the speed of the train and the aspect of the signals.

A simplified example of this representation, for an incident that will be analysed in the next section, is shown in fig. 2. Each box represents a block of rail line, and each block ends with a signal (protecting the following block). Each box of the S row shows the status of the signal that is placed at the end of the corresponding block. The signal status is represented using grey nuances: black stands for red signal, medium grey means yellow and light grey means green. Thus, a light grey box means that the signal at the end of the associated block was green, a box with two colours, for example medium grey and light grey, means that the signal was yellow when the train
entered the block and then switched to green while the train was running through the block. Each of the first four rows is dedicated to one of the main resources involved in the event. But there are several minor resources involved in the incident, that have not been reported to keep the example simple. For the same reason only two interactions are shown with arrows: the one between train driver (TD) and train course form (TCF) and the one between the train driver and the signal repetition system (SR). The other interactions are intuitive in this simple example. The train started from station A at 7:29, passed several signals, including the red protection signal at the entrance of station B, and stopped with an emergency break at 7:36. The speed of the train, between the two stations, till the SPAD, is shown graphically in the fifth row.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TCF</td>
<td>TD</td>
<td>SR</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2.** Graphic representation of the SPAD conditions

**Identification of Critical Issues and Focus Group.** A preliminary list of the critical issues is derived from the analysis of the discrepancies between theoretical and real interactions between resources, using a set of checklists. Then the next step consists in collecting supplementary information about the critical issues identified through an investigation on the spot where part of the incident is simulated and video-recorded. We noticed that in railways, where most of the processes are usually slightly structured, there is not a univocal interpretation of the activity of a process. Different railway workers have different views and adopt different procedures for the same process. In addition, quite often, they have difficulties in describing formally their behaviours that are almost automatic, because developed along the years. Video-recording helps in understanding and reviewing the activity with the involved workers and facilitate the identification of automatic actions, that are sometime extremely important for the comprehension of the incidental events. (A very good example of a wrong automatic action, during the interaction with a computer-based system, will be
reported in the next Section). Testimonies, rules and procedures, technical documentation, relevant scientific reports and standards are other possible sources for the supplementary information. Critical issues, together with the approaches used to manage and control them, are then analysed and discussed in a focus group. The focus group is a moderated and stimulated discussion between selected participants [10]. It must involve at least one representative for each of the professional roles involved in the SPAD, because they are the stakeholders of the knowledge required for the related processes. There is no need to have the same persons who were directly involved in the SPAD under investigation, on the contrary this is counter-effective because of the related strong emotional bias.

Analysis of Critical Issues and Identification of Possible Remedial Actions. The last phase of the methodology concerns the identification of the remedial actions. This is done within the focus group where proposals to remove or mitigate the critical issues are discussed. The historical data-base can support the cost-benefit analysis of the proposed remedial actions, by analysing the number of past incidents that would have been positively affected by these actions. Some critical issues can remain open and be the subject of additional analysis, to get additional information and facilitate the definition of possible remedial actions.

3 Application of the Methodology in a SPAD Investigation

The methodology described in the previous Section has been validated by investigating retrospectively three SPADs that happened in the Italian railways between April '97 and November '98. Of particular relevance is an incident, described in the following, where the usage of a computer based equipment, in a way that was quite different from the one hypothesised by the system designers, contributed to the SPAD. The investigation has been simplified to make it more clear and understandable. For example the analysis and modelling of the process were complicated by the presence of two drivers (this is still usual in the Italian railways), while we report here, in models and graphical representations, just the role and interactions of the main driver. This Section describes only the aspects of the incident analysis concerning the computer based system, but we want to emphasise that this is a classical case where the final event is the outcome of a combination of a large number of latent and active failures. A full report of the investigation is available in [11].

The train left the station of A at 7:29 with a yellow signal and increased its speed passing several green signals. Then it passed a yellow signal and a red one at the entrance of station B, called protection signal of station B. The driver did not decrease the speed until he realised the train was entering the station, then he stopped with an emergency break at 7:36. The incident had no consequences in terms of physical damages or injuries to humans but at a preliminary analysis it sounded uncommon. The environmental conditions were perfect, with good weather and good visibility. The line signal was visible well before the minimal distance required by the applicable rules. The two drivers were not tired, they had just started their service, and had not physical problems. In addition, they had a good driving experience without any relevant problems in the past. Above all, the train was equipped with a perfectly working Signal Repetition System. The Signal Repetition System controls the speed
of the train and compares it with the maximum speed compatible with the characteristics of the line. In addition, it "anticipates" within the cabin the status of the signal the train is going to encounter, providing audible and lighting warnings for green signals and other than green signals. If the signal is not green the driver has to acknowledge the audible and lighting warnings within a predefined time. In case the driver fails to acknowledge a signal within the required time the system applies an automatic emergency break. The interface of this system, used to acknowledge the signals is shown in fig. 3.

![Fig. 3. The Signal Repetition System](image)

The investigation started with the modelling of the processes. A simplified version of the one concerning the signal crossing is reported in fig.1. An electric signal provides the information to the Signal Repetition System about the status of the coming signal (first row in fig. 1). The Signal Repetition System provides the same information to the train driver (first row in fig. 1) who has to acknowledge in case of a signal other than green (second row). If the driver does not acknowledge the Signal Repetition System will activate the braking system (second row). Then the signal provides the status information to the driver (third row). Should the real procedure follow this model and the related interactions between resources, SPADs would never happen. But, the analysis of the historical data base of the Italian railways evidenced that about 10% of the SPADs of the last two years affected trains with a correctly working Signal Repetition System [11]. Also the available literature confirms the problem with similar systems. Hall, in an accurate survey on UK railway safety covering the period 84-97 [3], reports some events affecting trains equipped with working Automatic Warning Systems (a system with the same functions and architecture of the Italian Signal Repetition System).

A good insight came from a normal driving session, conducted with video-recording during the investigation. The session evidenced the real way experienced train drivers use this equipment in operation, identifying the most likely reason of the inefficacy of the Signal Repetition System in the case under investigation. A quite common condition is that a signal, placed at the end of a block, is yellow when the train enters the block and then switches to green while the train is advancing along the block. Then, the Signal Repetition System anticipates the yellow when the train enters the block and the driver has to acknowledge this yellow status, but he does not have to take any actions when the train arrives at the signal because in the meanwhile it
switched to green. This happened twice in the case under investigation, as shown in the first row of fig. 2. Thus, experienced drivers do not perceive the system as a support to the driving activity rather than as an inevitable disturbance, starting when a block is entered, that must be silenced as soon as possible. They stand in front of the windscreen looking outside, with a finger on the acknowledge button of the Signal Repetition System, ready to push it as they enter a new block and the "noisy warning" is going to start.

<table>
<thead>
<tr>
<th>Theoretical resources interaction for the process: Signal Crossing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Train position</strong></td>
</tr>
<tr>
<td>Train driver</td>
</tr>
<tr>
<td><strong>Entering the block</strong></td>
</tr>
<tr>
<td><strong>Running through the block</strong></td>
</tr>
<tr>
<td><strong>Approaching the signal</strong></td>
</tr>
</tbody>
</table>

Fig. 4. Theoretical resources interactions

They acknowledge automatically and so quickly that they cannot perceive if the light indicating a red or a yellow signal is on, losing the essential information provided by the Signal Repetition System. This type of interaction of the drivers with the interface is quite common and several drivers confirmed this is the behaviour they usually adopt. But this was an automatic habit, and only reviewing the activity with the video recording within the focus group were they able to identify the potential critical issue associated with it.

The differences between the theoretical and the real resources interactions are shown in fig. 4 and 5, through an instantiation, to the particular case under investigation, of the process described in fig. 1. The driver behaviour is worsened by the interface and the physical layout of the system. The system interface is not designed properly because it requires the same type of action to acknowledge different system conditions: yellow and red signals are both acknowledged by driver pushing to the same button, thus the interface doesn't ensure an adequate referential distance to different indications [12]. In addition the different lights, used to repeat the different signal statuses, are of the same size and the physical layout of the system within the train cabin does not allow the second driver to see which light is on. The
training procedure for train drivers provide recommendations for the usage of the system but it is not verified in practice if these recommendations are adequate and applicable.

<table>
<thead>
<tr>
<th>Real resources interaction for the process: Signal Crossing</th>
<th>Main resources involved in the process</th>
<th>Knowledge owned by the driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train position</td>
<td>Train driver</td>
<td>Acknowledge entering the block without identifying the on light</td>
</tr>
<tr>
<td></td>
<td>Signal repetition system</td>
<td>Warns operator &amp; immediately returns to normal status</td>
</tr>
<tr>
<td></td>
<td>Signal</td>
<td>Send status information</td>
</tr>
<tr>
<td></td>
<td>Driver has an expectation based on the most likely signal status that could be non realistic</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. Real resources interactions

When the system was installed it was validated assuming the operative usage shown in fig. 4. With this operative usage the system demonstrated effectively its potential support for preventing possible SPADs, and satisfied the customer requirements. But, the real operative usage, shown in fig. 5, is quite different from the one hypothesized by the system designers. The adoption of this new and error prone operative usage is due to several reasons. Only part of them are attributable directly to the train drivers. Other reasons include the described design errors, inadequate training procedures and bad physical layout of the system.

4 Conclusions

The paper described a methodology developed to investigate SPADs and promote reactive actions for the removal of latent failures. The methodology has been validated by investigating retrospectively three SPADs that happened in the Italian railways. It was very effective in detecting problems at the interactions between the resources that contribute to the driving process, in particular between human and computer based systems. The strong involvement of the users of the interactive systems was a key element to identify the real operative usage of the system, and this usage was quite different from the one hypothesised by the system designers. Future research directions are the extension of the methodology for the analysis of different events, and a more direct support in the identification and evaluation of possible corrective actions for the removal of the latent failure conditions.
References

Experimental Evaluation
of Fault Handling Mechanisms

Piotr Gawkowski and Janusz Sosnowski

Institute of Computer Science, Warsaw University of Technology,
ul. Nowowiejska 15/19, Warsaw 00-665, Poland
{gawkowsk, jss}@ii.pw.edu.pl, fax: +48 22 8251635

Abstract. The paper deals with the problem of enhancing fault handling
capabilities (detection, masking, recovery) in COTS based systems by means of
various software techniques. The effectiveness of this approach is analyzed
using fault insertion testbed. Discussing test results, we concentrate on fault
propagation effects and experiment tuning issues.

1 Introduction

High requirements for performance and reliability appear in many applications. So,
effective solutions to these problems are needed. In the literature many techniques of
fault tolerance based on specially developed hardware have been described (e.g. [10]).
Most of them are quite expensive and difficult to be implemented with COTS
elements. To reduce costs and profit from recent technology advances, it is reasonable
to rely on COTS elements and systems (much cheaper and more reliable than
specially developed circuitry - due to the matured technology and the experience
gained in many applications). To increase fault handling capabilities, we can use
various system hardening mechanisms implemented in software and supported with
simple hardware circuitry. Several ideas have been described in the literature such as
N-version programming, recovery blocks, program structure replications, control flow
checking, assertions, recalculation etc. ([1,2,3,10,11] and references). An important
issue is to evaluate the effectiveness of these mechanisms.

In the paper we concentrate on the problem of limiting fault sensitivity of COTS
based systems with software approaches. This is especially important for industrial
embedded systems. The main contribution of the paper is experimental methodology
of evaluating fault handling capabilities in real systems. For this purpose we use
software implemented fault inserter FITS which has been developed in our institute
[13]. As compared with other similar tools (e.g. [2,4,6,9] and references) it assures the
capability of detailed tracing fault effects in the system. Moreover the knowledge of
fault inserter implementation is quite useful in result interpretation or developing new
functions. In our experiments we were mostly interested in checking transient faults
effects due to their increasing dominance in practice [2,12]. Moreover, mechanisms
dealing with transient faults cover in a large extent also permanent faults. The
obtained results showed that some general opinions on classical mechanisms are not
accurate. The presented methodology can also be applied for other mechanisms. It
facilitates the designer to find weak points and to improve the effectiveness of these mechanisms.

2 Fault Handling Mechanisms

To make the system fault resistant we have to use various fault detection and fault tolerance mechanisms. They operate either on the basis of fault masking (e.g. voting, error correction codes) or detection and error recovery. Depending upon the fault (permanent, intermittent or transient), various techniques should be used. The most important thing is to detect faults. In recent COTS microprocessors some fault detection mechanisms are embedded, they relate to simple checking of on-chip cache RAM and bus operation (parity codes), as well as more sophisticated detectors such as: access violation, array bounds exceeded, data misalignment, stack overflow, illegal instruction, privileged instruction, integer overflow, integer divide by zero, stack overflow, FPU exceptions.

An important thing is to discriminate transient from intermittent and permanent faults. Transient faults are caused by various external and internal disturbances (electrical, cosmic and ion radiation etc.) they appear randomly and their effects can be eliminated by recalculation etc. Permanent and intermittent faults relate to circuit defects and can be masked out by circuit redundancy (massive or natural in COTS elements). The crucial point is to distinguish these two classes of faults: the overwhelming majority of the occurring faults belongs to transient category. Algorithms dealing with this problem have been proposed in the literature [3,12]. In our projects we rely on count and threshold scheme from [2]. In the single threshold count algorithm (STC), as time goes on, detected faults for a specified component are counted with decreasing weighs as they get older, to decide the point in time when keeping a system component on-line is no longer beneficial. Here we can use a simple filtering function:

\[ \alpha^L = \alpha^{L-1} \cdot K \quad \text{if} \quad J^L = 0 \] \[ \alpha^L = \alpha^{L-1} + 1, \quad 0 \leq K \leq 1 \] (1)

where \( \alpha \) is a score associated to each not-yet removed component to record information about failures experienced by that component, \( J^L \) denotes L-th signal specifying detected fault (1) or lack of detected fault (0) in time moment L. Parameter K and threshold \( \alpha_t \) are tuned so as to find the minimum number of consecutive faults sufficient to consider a component as faulty (permanent, intermittent). Similar filtering function can be formulated as an additive expression. In more complex double threshold count scheme (DTC) we have two thresholds \( \alpha_t \) and \( \alpha_{hi} \). Components with \( \alpha \) score below \( \alpha_t \) are considered as non-faulty (disturbed or no by transient faults), above \( \alpha_{hi} \) as faulty (permanent or intermittent fault) and in between \([\alpha_t, \alpha_{hi}]\) as suspected (further decision can be based on extensive testing).

In COTS systems hardware error detectors can be enhanced with software procedures such as assertions, which allow us to check the correctness of the obtained results by verifying some properties e.g. monotonicity of ordered set of items, correct ranges of results as compared with previous results (for many problems inverse operation can be used). In general this approach is application dependent and not always acceptable. For some applications checksums are quite effective. For example
multiplying two matrixes with additional checksum row and column we can detect error by checking result checksums. Moreover this technique is considered as single fault tolerant [10]. Our experiments showed that this property is significantly limited for software implementation. For some applications verification of program control flow may be satisfactory (some microcontrollers). Here we can use algorithms from [1,8] which are based on inserting some tags into program and then checking if they are followed (during the execution) in the appropriate order.

Quite good results can be achieved by recalculations performed with the same code and data or with different codes, relocated, swapped or shifted data (to deal with permanent and intermittent faults). This approach may use comparison of two results (only error detection) or voting on three or more results (fault masking). The granularity of the recalculation can be fine (each procedure verified separately) or coarse (final result verification). Some kind of fine-grained checking based on comparison was proposed in [2,11]. The authors developed rules of transforming a primary program into redundant structure by duplication. These rules are as follows:

- **#1:** Every variable $x$ must be duplicated: let $x1$ and $x2$ be the names of the two copies,
- **#2:** Every write operation performed on $x$ must be performed on $x1$ and $x2$,
- **#3:** After each read operation on $x$, the two copies $x1$ and $x2$ must be checked for consistency, and an error detection procedure should be activated if an inconsistency is detected,
- **#4:** An integer value $k$, is associated with every basic block (branch-free) $i$ in the code,
- **#5:** A global execution check flag ($ecf$) variable is defined; a statement assigning to $ecf$ the value of $k_i$ is introduced at the very beginning of every basic block $i$; a test on the value of $ecf$ is also introduced at the end of the basic block,
- **#6:** For every test statement the test is repeated at the beginning of the target basic block of both the true and (possible) false clause. If the two versions of the test (the original and the newly introduced) produce different results, an error is signaled,
- **#7:** An integer value $k_j$ is associated with any procedure $j$ in the code,
- **#8:** Immediately before every return statement of the procedure, the value $k_j$ is assigned to $ecf$; a test on the value of $ecf$ is also introduced after any call to the procedure.

In many applications combining hardware and software approaches to fault detection and tolerance is effective. This holds not only for sophisticated high dependability systems [7,10] but also for simple systems oriented towards automotive applications, banking, process control and various security systems. In many of these applications it is important to assure high degree of safety (low probability of dangerous states).

Developing software procedures enhancing system fault resistivity we should be conscious that these procedures are also susceptible to faults. Hence it is reasonable to check these procedures in fault insertion experiments. This is especially important if these procedures (e.g. comparison, voting, checksum or code verification) are inserted frequently into the primary program (fine granularity). We have analysed the susceptibility to faults for many programs with various fault handling mechanisms as well as these mechanisms separately. For this purpose we use fault insertion testbed FITS [13] described in the sequel. Some representative experimental results are given in section 4.
3 Fault Insertion Testbed

Faults to be inserted (using FITS) are specified on logical level as disturbances of processor registers, code and memory locations. The following fault types are possible: bit inversion, bit setting, bit resetting, bridging (logical AND or OR of coupled bits). It is also possible to select pseudorandom generation of fault types (single, multiple). Duration of the fault is specified in number of instructions for which the fault must be active starting from the triggering moment. This mechanism gives the possibility of setting transient and permanent faults. The assured fault specifications make it possible to model physical faults in various functional blocks of the system, in particular, processor sequencer, processor ALUs, FPU, general purpose and control registers, bus control unit, RAM memory. FITS provides high flexibility in specifying the moment of fault injection - fault triggering point. The set of faults to be inserted can be specified explicitly or generated in a pseudorandom way.

For each test (fault injection) FITS sets trap in appropriate triggering point within the analyzed program. During the test execution FITS takes over the control after the trap, performs fault insertion (e.g. by changing register state, instruction code or memory location) and traces the target application execution for a specified number of instructions. The exit code and all generated events, exceptions and other result data are registered in result file and database. In particular case, the exit code can be defined as a final result. For some applications it is useful to add special subroutine checking the correctness of their execution (they can take into account files generated by the analyzed program etc.). This subroutine is specified as DLL file (external to FITS). In general we distinguish 4 classes of test results: C - correct result, INC - incorrect result, S - fault detected by the system (FITS delivers the number and types of collected exceptions), T - time-out. If the analyzed program generates user defined messages (e.g. signaling incorrect result), they are also monitored by FITS and specified in the final test report (U).

As the perturbed bit and the instance of occurrence of the upset are known, we can trace the propagation of the injected fault from the moment of its occurrence to the end of the program or the activation of some error detection mechanisms etc. This allows us to explain many system behaviours. Many faults may have no effect on system operation (i.e. no error will be activated). This may result from three reasons:

- **non activated fault** generated in a location which is not used or the fault effect is overwritten with system operation, fault state consistent with the value needed during normal operation etc.,
- **masking effects** due to hardware redundancy (e.g. error correction codes),
- **algorithm robustness** - algorithm natural redundancy (e.g. writing specified pattern into a table with a loop is performed correctly despite a fault decreasing current loop index).

Hence, an important issue is to deal with faults influencing system operation. This was taken into account in our experiments (section 4).
4 Experimental Results

We have performed many experiments with faults injected into various application programs executed in Windows environment. The aim of these experiments was to check natural capabilities of COTS systems in detecting and tolerating faults and the improvement of these capabilities achieved with software procedures. In addition, we analysed fault resistivity of error handling procedures. In all these experiments an important issue was the problem of getting representative results, taking into account various input data, activity of used system resources etc. This is illustrated in some selected examples.

**Enhancing Fault Handling Capabilities.** Improving fault resistance to faults with software approaches is illustrated for sorting program BubbleSort implemented in three versions:

- **V1** - basic version without any fault detection capability,
- **V2** - version V1 modified according to rules #1-#8 (section 2),
- **V3** - version with simple assertions as the only fault detection mechanism. The sum of input vector was computed before sorting. That sum was checked after sorting. Sorted vector was also checked for monotonicity.

All three versions were compiled with Microsoft Visual C++ 6.0 compiler. Injected faults were random single bit flips. Faults were injected into processor registers (R), data area (D) and instruction code (I) of the considered version. Table 1 summarizes test results in percents (see section 3) for 5 different input data sets (over 20000 faults injected pseudorandomly for each version).

<table>
<thead>
<tr>
<th>Version</th>
<th>C</th>
<th>INC</th>
<th>S</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td>62.8-64.7 %</td>
<td>1.6-3.5 %</td>
<td>33.3-33.8 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>V2</td>
<td>62.1-64 %</td>
<td>0.8-1.3 %</td>
<td>31.9-32.9 %</td>
<td>0 %</td>
<td>2.2-4.2 %</td>
</tr>
<tr>
<td>V3</td>
<td>62-65 %</td>
<td>0.1 %</td>
<td>31-34 %</td>
<td>0 %</td>
<td>2.9-3.8 %</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td>58-64 %</td>
<td>30-35 %</td>
<td>4.7-5.8 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>V2</td>
<td>51-53 %</td>
<td>16-39 %</td>
<td>4.2-4.3 %</td>
<td>0 %</td>
<td>4.2-28.3 %</td>
</tr>
<tr>
<td>V3</td>
<td>51-57 %</td>
<td>1.4-5.9 %</td>
<td>4.2-4.7 %</td>
<td>0 %</td>
<td>32-42 %</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td>6.5-24.9 %</td>
<td>11-26 %</td>
<td>59-64 %</td>
<td>2.6-4.4 %</td>
<td>0 %</td>
</tr>
<tr>
<td>V2</td>
<td>14-27 %</td>
<td>3.1-10.9 %</td>
<td>51-57 %</td>
<td>0.2-0.3 %</td>
<td>17-20 %</td>
</tr>
<tr>
<td>V3</td>
<td>7-17 %</td>
<td>0-0.2 %</td>
<td>58-63 %</td>
<td>2.9-5.4 %</td>
<td>18-26 %</td>
</tr>
</tbody>
</table>

We can observe that simple assertions are the most effective. Unfortunately, this approach is application dependent. Code transformation solution is universal; however, it leads to significant time and RAM space overhead. In tested programs the ratio of the number of machine instructions executed in case of V2 program to the number of machine instructions executed by V1 program was in the range 2.74 to 4.07 depending upon input vector - for V3 this ratio was 1.12-2.01 (time overhead). Size of code was around 3 times higher in V2 and 1.35 for V3. Program V2 used 107 different data cells compared to 53 used by V1 and 55 in V3 program.
Data size increase in case of V2 program relates to doubling all data variables and ecf variable. Code and execution time overhead strongly depend on algorithm specificity. For programs with wider branch free blocks the overhead should be lower. Transformation rules decrease the percentage of timed-outs in case of faults in instructions. This results from on-line control flow checking embedded in transformation rules. To better understand the weaknesses of transformed code some special experiments were performed. We took a closer look on faults in EAX register. For that purpose enhanced version compiled with additional debug information was used. Hence we were able to identify operations on the source level that were sensitive (incorrect result) to faults injected into EAX register. Let’s consider the following expression, which is performed after each read operation on variables size1 / size2 – according to the transformation rule #3:

\[
\text{If (size1 != size2) Error(...)};
\]

Variables size1 and size2 are two copies of size variable in the standard application (transformation rule #1). At the machine code level this comparison operation was performed with the use of EAX register (one of the variables was stored intermittently in EAX). Because of that fault in that register during comparison can activate oversensitive error detection procedure. It’s worth noting that fault injected does not affect any primary variable used in that operation. So many checking operations may create oversensitive reactions.

More dangerous situation is returning incorrect result without signaling fault detection. In the considered application this situation took place when fault was injected during execution of a procedure exchanging two elements of the sorted table. For that purpose temporary variable swapArea (swapArea1 and swapArea2 respectively in enhanced version) was used. During calculations the variable replica was temporarily mapped into EAX register; fault injected into the EAX leads to writing incorrect value into result table. This fault is not detected because the considered table element is not read during further program execution. The proposed rules (section 2) check variable during reading - do not verify results after writing. Hence, we propose additional checking – not only on source data but also on destination variables.

Another dangerous effect appears when test statement does not contain else block. If a fault affects program’s control flow - not test operation arguments – an absence of additional testing of control flow causes lack of fault detection. In rule #6 it is not clearly stated if else block is obligatory. For better fault detection capability else block should be mandatory with additional testing as specified in transformation rule #6.

Similar problems affect for, while and do..while statements. It’s worth noting that for some applications and input data vectors this may not hold because of algorithm and input data specificity. To improve fault detection capabilities we suggest to place additional checking on exiting condition of these blocks. Unfortunately, this is not a simple transformation rule because these blocks can be exited at any point. Extra checking code should take into account all conditions for leaving these blocks.

We have observed that control flow checking organized on the source code level is not good enough to cover all control flow errors at the corresponding machine code
level and unfortunately gives very high overhead in terms of code size and computation time. The advantage of such solution is its generality – independence from the algorithm of the transformed program.

It is worth noting that the same algorithm may have different fault sensitivity to faults depending upon the used compiler. For Qsort program in case of faults in effective addresses we obtained around 5% of incorrect results for version compiled with Ada95 and 50% for version compiled with Microsoft Visual C++ compiler. Opposite results were obtained in experiments with faults injected into program’s code: 26% of tests were incorrect in Ada version while only 14% in VC++ version. Sensitivity to faults in processor registers was similar for all versions.

In similar way we analysed other programs. Multiplying two matrixes with checksums in columns and rows reduced the percentage of incorrect tests from over 62% (for the basic version) to 0.5% in case of faults injected into data area. The correct results constituted about 33% in both cases (for faults injected into the code about 3%). This was improved to 52% by adding recalculation (and to 87% for faults in code). Recalculation increased also percentage of correct results in case of faults injected into processor registers from 2.8% for basic version and 4.7% for version with checksums to over 86%.

In basic version of the program for matrix multiplication activity ratio (AR) of register EAX (percentage of time during which register holds value for further use) is 56%. This strongly relates to percentage of correct results in experiment (C=44%). In case of EBX register AR=92% (C=10%) while in case of ESI register AR=41% (C=61%). A difference between AR and 1-C show natural code robustness and this is also strongly related to the input data used in experiment.

In another experiment we analysed calculation oriented application with fine and coarse grained voting on three items. Inserting faults into the code we obtained C=95.6% and C=58% of correct results for coarse and fine grained voting respectively (incorrect results INC=0.7% and INC=10.5%). Faults injected into data area resulted in INC=2.8% (C=95.6%) and INC=5.7% (C=92.5%) respectively. For faults injected into processor registers C=83%, INC=0.8% and C=71%, INC=4.9%. Fine grained voting is more susceptible to faults due to higher percentage of voting code in the whole application.

**Checking Error-handling Procedures.** Using various special software procedures for detection and error handling we have to be conscious that these procedures are susceptible to faults. So, an important issue is to check system behavior in such situations. We illustrate this for STC and DTC algorithms described in section 2.

Checking fault resistivity of error handling procedure based on single and double threshold algorithms we have generated several fault scenarios i.e. the distribution in time of fault detection signals. These scenarios are stored in files, which are treated as inputs to the programs implementing the considered algorithms. For each of the generated scenarios we analysed non disturbed algorithm behaviour (to find golden run) and then performed fault insertions during algorithm execution. In fig 1 we give examples of $\alpha$ values in the function of time (standardised units) for different input scenarios and different values of parameters: $K$, $\alpha_a$ and $\alpha_{hi}$. 

Results of fault insertion experiments are given in tab. 2 (for both algorithms). Faults were inserted in processor registers (R), data area (D - input data and program variables), program code (I) and floating point unit register stack (FPU). All experiments involved many pseudorandomly generated faults. For each group of faults we used 4 different input data scenarios. The results show the percentage of incorrect (INC), correct (C) outcome of the program as well as the percentage of faults detected by the system (S - includes time-outs which appeared only in cases denoted in bold - they contributed 0.1-0.4%). It is worth noting that only faults inserted into program and registers were detected by the system. The most critical situation relates to wrong error classification i.e. incorrect results (INC). Here we observe relatively high fluctuation depending upon inputs. Especially for faults inserted into data area (0.1-41.4% for STC and 0.2-24.9% for DTC algorithms). The lower percentage of INC states related to input scenarios with $\alpha$ scores uniquely assuming values significantly below or over the specified thresholds e.g. scenarios A3, A4. Scenario A1 relates to monotonic increase of $\alpha$ exceeding slightly threshold $\alpha_T=10$ in the last phase. Scenario A2 was similar to A1 except that $\alpha$ did not cross the threshold. Test set B1 relates to monotonic increase of $\alpha$ for most of the time within area $\alpha_L$, $\alpha_H$ (but not exceeding $\alpha_H$ - suspected component). Test set B2 is similar except that finally $\alpha$ exceeds slightly $\alpha_H$ (faulty component). The analyses showed low sensitivity to FPU faults (0.1-2.3%). This resulted from the fact that FPU
instructions contributed 38% of the executed code and used mostly only 1 or 2 floating point unit registers (faults were injected into FPU registers).

Table 2. Results from fault insertion experiments for STC and DTC algorithms (data sets 1,2,3 and 4 correspond to scenarios A1-A4 and B1-B4 of STC and DTC algorithm, respectively)

<table>
<thead>
<tr>
<th>Fault loc.</th>
<th>Data set</th>
<th>Algorithm STC</th>
<th>Algorithm DTC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>INC</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>61.6 %</td>
<td>10.4 %</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>52.8 %</td>
<td>18.6 %</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>67.6 %</td>
<td>6.8 %</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>55.0 %</td>
<td>16.4 %</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>98.7 %</td>
<td>1.3 %</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>58.6 %</td>
<td>41.4 %</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>95.2 %</td>
<td>4.2 %</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>99.9 %</td>
<td>0.1 %</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>22.0 %</td>
<td>28.5 %</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>35.4 %</td>
<td>18.7 %</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>23.3 %</td>
<td>31.8 %</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>37.1 %</td>
<td>17.1 %</td>
</tr>
<tr>
<td>FPU</td>
<td>1</td>
<td>97.6 %</td>
<td>1.8 %</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>99.4 %</td>
<td>0.1 %</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>95.4 %</td>
<td>2.3 %</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>99.4 %</td>
<td>0.1 %</td>
</tr>
</tbody>
</table>

In a similar way we analysed other error handling procedures e.g. signature analyser based on CRC encoder and software implemented voting. Injecting faults into CRC encoder data area we obtained INC=52% (incorrectly computed signature) and C=45% (correct). For the enhanced CRC encoder (triplication of all variables, consistency checking, simple checkpoint saving inside computation loop and rollbacks in case of inconsistency detection) significant improvement has been achieved: C=98.3%, INC=0.01% and S=1.6%. For faults injected into registers we obtained INC=14.9%, C=44% and INC=0.4%, C=63% for the basic and enhanced version respectively. Faults injected into program code resulted in INC=38.7% (C=8.9%) and INC=2.4% (C=42.2%).

Injecting faults into voter data area, 100% of correct results were obtained. Faults in instruction code resulted in INC=5% of tests (C=55%, S=40%). For faults injected into processor registers we obtained INC=1.5%, C=67% and S=31.5%.

5 Conclusion

The paper showed that COTS based systems have some capabilities in detecting and masking fault effects. These capabilities can be significantly improved with special software procedures. An important issue is to verify the effectiveness of these procedures. This can be performed with fault insertion experiments. Designing such
experiments, we have to take into account the problem of selecting representative inputs for the analysed application, most sensitive registers etc. Moreover, better interpretation of experiments needs correlation of the obtained results with resource activity etc. FITS gives us such information. So we can avoid disturbing not used registers, correlate low incorrect result percentage with low resource activity etc. Appropriate tuning of performed experiments is of great importance. Well organised experiments allow the designer to get an insight in the consequences of faults and the efficiency of the detection/correction mechanisms. Moreover, this is the basis for developing sophisticated analytical models (e.g. [5]).

Acknowledgment. This work was supported by Polish Scientific Committee grant no. 8T11C 020 16.

References

3. Bondavalli, A., et al.: Threshold Based Mechanisms to Discriminate Transient from Intermittent Faults, IEI:B4-17-06-98, IEI - Pisa
The COTS Debate in Perspective

Felix Redmill
22 Onslow Gardens
London N10 3JU
UK
Felix.Redmill@ncl.ac.uk

Abstract. Safety standards require assessment of development-process evidence for all parts of safety-related systems. In spite of this, there is a move to use commercial off-the-shelf (COTS) components in safety-related systems, and a usual feature of COTS products is a lack of such evidence. There is therefore a debate as to the appropriateness of COTS products in such applications. This paper discusses not only evidence, but also risk, the other issue at the heart of the COTS debate. It also addresses the other side of the debate: a challenge to the rigorous requirements of the standards. Finally, the paper proposes a convention on the evidence that should be provided to support claims for the safety of COTS items.

Key words: commercial off-the-shelf (COTS) products, evidence, risk, safety standards.

1 Introduction - Current Safety Thinking

Developers of safety-critical systems are expected not only to achieve safety but also to demonstrate its achievement.

System safety must be achieved in the development processes. It cannot be added later but must be built in at the design and construction stages. It depends on the competence of the people who carry out these processes, the extent to which they understand their technologies and the system's environment, and the thoroughness of their risk analyses and their use of the results. Whilst standards emphasise the importance of the appropriate choice and management of processes throughout the development life cycle, it is also recognised that the use of some processes is considerably more difficult, lengthy and costly than that of others. A demand for all systems to be equally 'safe', or as safe as possible, would lead to many being unreasonably expensive. Thus, various methods of categorising safety levels have been introduced. For example, in IEC 61508 [1, 2] there is the scheme of 'safety integrity levels' (SILs) in which the need for greater risk reduction is represented by a higher SIL - on a scale of 1 to 4. It is accepted that the higher the SIL the higher must be the rigor of the techniques, tools and management processes during development, so these must be chosen to be commensurate with the SIL.

Although safety must be built in during development, there is a question about how it needs to be demonstrated. Certainly, evidence is required, and to ensure that the evidence of safety is tested, independent safety assessment must be carried out. Further, it is becoming common practice for a safety case to be drawn up, in which a
logical argument is developed to relate the evidence to specific claims of safety - and some standards (though not yet all) call for this. But where should the evidence be derived?

Because there are limits to the extent that software reliability can be determined by measurement [3], safety standards mainly fall back on the need for evidence of appropriate development processes. Yet, there are other possible sources of evidence, for example from experience of use and the results of testing.

At the same time that safety standards are mandating more difficult and expensive development processes, there is a move among system purchasers and suppliers to use commercial off-the-shelf (COTS) systems and components. Being developed for a wide market, their costs, at least at the time of sale, are likely to be considerably less than those of bespoke products. In addition, there are technological reasons to support the use of COTS products. For example, Dawkins and Riddle [4] point out that the relatively small safety-related systems market cannot sustain the rate of technological advancement stimulated by the huge commercial market, and that it would suffer technological retardation if it did not use COTS products.

But a feature of COTS products is that their suppliers do not usually provide evidence to facilitate their assessment for safety. In particular, evidence of the development processes, as demanded by the standards, is seldom made available - and without this the standards cannot be met. Yet, in response to this, COTS proponents would make two points:

- Evidence of 'good' development process does not guarantee safety;
- Evidence of the development process is not the only type of relevant evidence and that evidence about the product itself may be just as valuable.

A safety case for a system requires evidence for its credibility, but are the constraints imposed by the standards definitive, or do the COTS proponents have a point? There is a debate over whether the use of COTS products can be justified in safety-related applications and, if so, how. Central to this debate are two issues, evidence and risk.

There is also a question of whether the barrier to COTS - the requirements of the standards - is the problem. Are the standards too demanding at a time when COTS products are necessary in the safety-critical systems industry? Or are the proponents of COTS products really arguing that 'Cheapness Over-rides Threats to Safety'?

Modern safety standards are said to be goal-based - that is, they demand that their users define safety targets, but they are not prescriptive in defining the ways in which the targets should be met. Yet, the most influential standard, IEC 61508, defines which processes are appropriate to the different SILs. Claiming conformity with the standard is therefore constrained and can be expensive. It is also contentious, for there is no established link between the various processes and any product attributes. The COTS proponents can therefore claim, with some justification, that the restrictions placed by the standards are not based on a firm foundation.

It should be pointed out that COTS software is a sub-set of a larger class that might be labelled 're-used software'. Indeed, non-commercially acquired software that is to be reused is sometimes referred to as a NDI (non-development item) to distinguish it from COTS products. Software products not developed to bespoke standards, for whatever reason, are sometimes referred to as 'software of unknown pedigree' (SOUP). Software is increasingly being used in products that previously did not contain it, often without the knowledge and consent of the purchaser. Furthermore, it
is rare for 'legacy' software to be accompanied by appropriate evidence. In this paper, the terms 'OTS product' and 'OTS software' are used to cover all these varieties of off-the-shelf items.

This paper offers an explanation of the COTS debate. It shows the arguments of both sides, considers the crucial issues of evidence and risk, and describes the case against the standards. It then calls for a convention on the evidence that should be provided in support of OTS products, and ends with a discussion of the issues.

2 The Issue of Evidence

The contentious issue is not one of COTS per se, but one of evidence. If OTS software were delivered with its development-process evidence, as required by the standards, as well as product details such as its source code, design, and test results, there would not be an issue. With sufficient evidence, a safety argument could be constructed to support its use (if its use were supportable) and assessment of its appropriateness in this or that application could be carried out. But, typically, OTS products are not accompanied by such evidence.

Evidence may be lacking for any of a number of reasons. In the worst case, a supplier may not use good engineering practice and may omit systematic documentation from the development process. In other cases, suppliers may justify confidentiality on the grounds that the required information is closely related to the product's success and therefore commercially sensitive. Then, in some cases there may be national security reasons for not including certain information with the exportation or dissemination of some products. Or, for legacy systems, the required information may have been destroyed or mislaid, it may not have been kept up-to-date when changes were made to the system, or it may never have existed.

While the lack of evidence is not a conclusive indicator that the product is inappropriate for an intended application, the following points are worth noting:

- If a product has been developed to the rigorous standards demanded by modern safety engineering, its supplier would often wish this to be known - though, at the same time, they may wish to protect their intellectual property rights.
- Safety is a system issue and context-specific. However good a COTS product may be, or however rigorously developed, it is only 'safe' or 'unsafe' insofar as it does not contribute to unsafe failures of the system in which it is a component, in the system's operational environment. Thus, objective demonstration of safety can only be retrospective. A safety case can therefore offer confidence and not proof, and it does this by coherent linking of the available evidence to an argument for safety. Bespoke systems, with substantial evidence of their development processes, are likely to be more convincing than OTS products with none, and arguments for their safety are more easily made and assessed.

Even high-quality software can lead to disaster when carelessly reused. Not only must there be confidence that OTS products are of the appropriate quality per se, the evidence must also exist to satisfy a safety assessor that they are suitable for the proposed applications. Because small changes in a system or its operating environment can lead to large changes in the effects of the behaviour of software, caution is needed when evaluating arguments that OTS products (or any software products) have been 'proven in use'.
On the other hand, the fact that certain processes were used in the development of a product is not proof that the product is appropriate for use in any given safety-critical system.

In the absence of development-process evidence, as called for by the standards, it may still be possible to acquire evidence by investigating the product itself. Frankis and Armstrong [5] suggest that to assess OTS software adequately, it must be validated against thirteen 'evidential requirements', and they list five classes of examination of the requirements: black-box methods, previous usage analysis, design-intention assessment, code assessment, and open-box assessment. The COTS debate is concerned with whether sufficient product evidence can be deduced to allow safety assessment, and whether such evidence can replace the process evidence demanded by the standards. These issues recur later in Section 5.

3 Potential Problems

At the point of sale, OTS products are likely to be cheaper than bespoke items. Yet, it is not certain that all the savings apply over the entire system life cycle, for there are some potential disadvantages to be considered. For example, if the OTS item is a black box, without source code and design details, the supplier must often be relied on for maintenance. This incurs financial costs and introduces a dependence that carries further implications. For example, security may require higher levels of staff, additional management structures, vetting procedures, and local arrangements.

Further, maintenance and other support is often only guaranteed if the system user accepts the frequent software upgrades developed by the supplier. Although a rapid upgrade path is often claimed as an advantage of OTS purchase, this can incur extra financial costs and have serious technical implications. For example, 'asynchronous' upgrading (e.g. when two suppliers make mutually incompatible upgrades) can create complex problems in the management of integrated systems, particularly when it is accompanied by the withdrawal of support for older versions.

A further problem derives from the possible composition of the upgrades, over which the purchaser will in most cases have no control. Because software is often viewed as easy to change, upgrades include not only new features and corrections to faults in previous versions, but also changes that might better have been achieved by other means. Thus, for the purpose of testing and safety assurance, an upgrade can seldom be perceived as 'changed' software, and should, rather, be considered as new. In addition, many of the new features, included to attract a wide market, are likely to be unwanted functions in the context of the safety-related application. They increase the volume and complexity of the total package and introduce the risk of undesirable and perhaps dangerous effects. Moreover, the functions in the OTS software that are to be used may require tailoring which, without the benefit of design and source-code documentation, could compromise safety.

Thus, not only at the time of purchase, but also at numerous other times throughout its life, the safety-related system will be dependent on untried software. At each of these points there is a lack of evidence of safety - and, perhaps, also a lack of safety. Further, the cost of carrying out safety assessments and re-assessments at all these times can be considerable. It should be estimated and allowed for during the initial system planning.
All this emphasises the importance both of resisting upgrades for as long as possible - at least until there is evidence that the upgrade has succeeded in non-critical use - and of keeping OTS, as well as bespoke software, under strict configuration control. Yet, it is not uncommon for user organisations to use OTS upgrades without question or trial and to exclude them from configuration control. Clearly, when safety is at issue, this is a dangerous practice.

A further consideration is that, if necessary evidence is absent, the process of negotiating the co-operation of the supplier and the assessors in the acceptance of substitute evidence could become protracted and costly - and this may occur not only at the initial purchase but also at the times of all subsequent upgrades. If the missing evidence is critical, there is also a risk that the safety case will not satisfy the assessors. Clearly, the risks of using OTS software should be assessed in detail at the safety-planning stage of a development project, at which time the assessors' requirements for evidence should also be elicited.

Thus, while there are forces pressing the designers of safety-related systems to employ OTS software, there are also factors which might negate its advantages and, in some cases, might cause it to be deemed unsuitable, even after a system has been developed. There are also severe technical limitations on the confidence that can be derived from the verification of 'black box' software (e.g. without source or design information), some of which are reviewed by Armstrong [6]. It is therefore not surprising that a great deal of research is currently enquiring into ways in which COTS software may be justified [e.g. 7, 8, 4].

This section has concentrated on issues that affect safety, but non-safety issues, such as commercial and security risks should also be taken into account in any cost-benefit analysis, over both the short and the long terms. The costs of extra risk management activities could negate the financial advantages of using OTS items. Furthermore, other system attributes such as reliability, availability, maintainability and security might be compromised by the need for more rigorous safety-risk management. And all these issues would add to the cost of the OTS product.

4 The Issue of Risk

The question of risk is as important as the question of evidence. If there were no risk attached to the use of COTS products, there would be no safety issue. On the other hand, there can never be zero risk. Even bespoke systems, developed to the highest standards, can and do fail. But how can we assess the risk of an OTS product without evidence?

For risk analysis, the nature of the required information about an OTS product depends on the part that the product is intended to play in the safety-related system. In most cases, the minimum requirement would be a thorough knowledge of its failure modes. Only then could the chains of cause and effect (derived using techniques such as FMEA (failure modes and effects analysis) and FTA (fault tree analysis)) leading from its failures to the system-level hazardous events be derived, as well as the potential consequences of each hazardous event.

When there is relevant evidence to support the integrity of the OTS product, risk analysis may address the likelihood of failure for each failure mode. Then risk values, based on the combination of likelihood and consequence, may be derived and tested.
for tolerability. Note, however, that confidence in the accuracy of the evidence is crucial, and that for software OTS products the evidence is unlikely to provide high confidence in estimates of the likelihood of failure.

If the OTS system is a black box, it is difficult to make a convincing argument for safety for a number of reasons. First, verification that all failure modes have been identified is not possible; particularly in the case of software, in which faults are systematic rather than random, previous experience cannot be assumed to have revealed them all. Second, failures monitored at the interface between the black box and the rest of the system cannot easily be traced to their true causes within the OTS system and cannot be assumed to be representative of particular failure modes. Further, fixes made at the interface may only address symptoms, leaving faults that could lead to dangerous system failures in the future.

Thus, in the absence of evidence to provide confidence in the reliability of the OTS product, it would be necessary to assume that if it could cause a hazardous event it would - i.e. that its probability of dangerous failure is unity. To do otherwise - certainly if it is software - would be contrary to safe practice. In their report on the failure of Ariane 5 Flight 501 [9], the Inquiry Board stated, 'software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.' Any attempt to assess the probability of failure would be purely speculative and risk analysis would have to be based on consequence alone.

A first step towards assessing the acceptability of a critical OTS product would then be to examine the tolerability of the consequence per se. But tolerability is itself a subjective notion, and involves a trade-off between safety and benefits. This gets to the heart of the COTS debate, for the proponents of the OTS product may argue that its benefits - such as cost, functionality, and immediate availability - outweigh the safety considerations. Just as it may not be possible to muster a convincing argument for safety, so it may not be possible to prove in advance that the OTS product is unsafe. Do we then apply the precautionary principle and adhere to safe practice? The commercial route may be attractive. Moral and ethical considerations enter the debate.

However, if a risk analysis is based on consequence, and if the consequences of failure are deemed intolerable, the next step should be to enquire into the use of a 'protection function' to reduce the probability of the consequence occurring. Such a protection function might be installed either in direct combination with the OTS product (as in Figure 1) or at some point in a fault tree between the product and the hazardous event. In determining the required probability of failure of the protection function and, thus, its safety integrity level according to IEC 61508 [1], no contribution to reliability could be assumed from the OTS component (its probability of failure is assumed to be unity).

A further point should be made here. The principle of Figure 1 is based on the assumption that the protection system is independent of what is being protected. Care should be taken in assuming that there are no modes of failure common to the protection function and the OTS component.

In summary, the requirements for evidence in support of the OTS product should be related to the risks involved. Early decisions on what is needed for assessment should be made in conjunction with the safety assessors. If adequate evidence is not available, risk analysis must be based on consequence, with decisions being required about the tolerability of the consequences of the various possible hazardous events,
and on whether and how the use of a protection function would justify the use of the COTS product.

The above discussion assumes that a risk analysis is carried out using whatever evidence is available. But suppose there is strong pressure to use the OTS product in the absence of necessary evidence, or without a protection function that analysis shows to be necessary, or, indeed, without analysis being carried out? How should decisions be made under such pressure?

From the safety perspective, the issue of employing a COTS product is one of deciding what risks to accept and when they are worth accepting. The more evidence there is, and the greater the confidence in it, the more consensus there is likely to be in decision-making. But when there is little or no evidence to support a claim for the safety of the COTS product, the decision becomes a gamble. Further, in the absence of evidence, there is no knowledge of the odds that pertain. Then, any decision will depend on who is making it and how they perceive the risks and benefits of using the COTS product. Value judgements are involved, and decisions need to be made not by one party but by consensus of all stakeholders.

### 5 Safety Standards – The Other Side of the Debate

As discussed above, one side of the 'COTS debate' is whether the use of OTS products in safety-related systems can be justified. The other side is a challenge to the standards and concerns the relevance of development-process evidence to safety assessment. The issue is this: it is the requirements of the standards that would preclude OTS products, so, if OTS products are in fact admissible, how can the standards' requirements for rigorous development processes be valid? After all, a good product can result from imperfect development processes. Moreover, the assumed relationship between product quality and the development processes has no proven foundation. Indeed, although there is good correlation between bad process and bad product, there seems to be poor correlation between good process and good product.

Further, the relationships, defined in standards, between safety targets and particular development techniques, are based on expert judgement only and are
therefore subject to question. Thus, if a safety-related system not developed according to the defined processes is found in practice to meet a given safety target, it may be claimed to refute the standards.

Most standards admit 'proven-in-use' evidence, though with constraints - for example, there should be no change in the software or its environment in the period during which the proof is gathered. If the in-use behaviour of the software, and the conditions under which it has operated, were monitored and recorded and are now shown to be relevant to the safety-related application, is the resulting evidence less valid than that of the development process? Many advocates of OTS products think not.

Moreover, OTS items not developed to the standards' rigorous processes (or without evidence that they have been) may also be subjected to testing, realistic operational trials, and simulated conditions. If the source code is available it may be inspected. Does such product evaluation outweigh the fact that the development processes may not have been in conformity with the requirements of the standards? The OTS product proponents think so. After all, a sound product can result from a flawed process. And it may be argued that a process not defined by the standards is not necessarily 'bad'.

Yet, the standards fall back on the development process with justification: as shown by Littlewood and Strigini [3], the extent to which the reliability of software (both OTS and bespoke) can be proved by testing is severely limited - not because appropriate tests cannot be devised, but because adequate testing cannot be carried out in cost-effective time. Thus, proponents of the standards argue that development-process evidence is an essential (though not always sufficient) part of any safety justification. There is also a 'precautionary principle' argument: if safety cannot be demonstrated, it should not be assumed - and this leads to the rejection of reliance on product evaluation because of the intractability of the task. Hence, OTS items must carry evidence that their development processes were as rigorous as those required by the standards for bespoke developments.

Yet, it is not only the task of product evaluation that is intractable. Devising a development process which is 'perfect' for the intended application, and managing it perfectly, are impossible demands. So perhaps the OTS proponents have a point.

Further, although the appeal of standards to the development process has technical justification, the lobby for OTS products is not based on cost alone and still challenges the validity of the approach. The failure of software development projects is legendary, and even those employing the most rigorous processes have frequently not only gone over-budget and over-time, but also produced systems that have been shown at an early stage of testing or operation to be unsuitable. Rigorous processes are considered cumbersome and unable, of themselves, to 'deliver the goods'. Thus, there is a view that reliance on such processes to achieve safety could be misplaced. Simpler, faster, less costly methods are sought, and these are perceived to exist in the commercial market where intense competition keeps costs low and propels technological advances that are quickly introduced into products.

So, are successful previous use and product evaluation sufficient evidence on which to base safety-related application of an OTS item? Not necessarily. We would need to carry out a great deal of comparison in order to claim that the old operational situation is representative of the new. We also know that black box testing is intractable and cannot wholly prove the product. Moreover, it is not uncommon for a software fault to cause a failure for the first time after years of operation, simply
because the relevant logical path had never previously been exercised. Thus, the constraints of the standards on using a 'proven-in-use' argument are not without justification.

Yet, when we enquire into the historical basis for the appropriateness of development-process evidence, in both the success of projects and the quality of bespoke systems, we find little to inspire confidence. Further, although the standards advise on what to do in order to achieve safety, they do not yet offer guidance on what must be demonstrated in order to make safety claims. Thus, the debate continues, and it is healthy for the safety-critical systems industry that it should.

6 We Need a Convention for the Provision of Evidence

It is typical to assume that COTS implies a lack of evidence and that suppliers do not want to provide evidence to support a safety argument. But do we have to accept these assumptions? Some suppliers (for example, of operating systems) are keen to be respected in the safety-related systems community and would provide evidence if asked to do so. This willingness should be encouraged, for there will be no evidence if we do not call for it.

So, might we strive to develop the terms of a convention for the provision of relevant evidence? There are a number of categories of evidence that could be made available.

First, there is direct evidence about the OTS product itself, for example the test plans, cases and results, and information on experience of its use. Second, there is evidence of the development process, for example, to demonstrate that it complies with good practice or with some accepted standard. And third, there is evidence to give confidence in the company that developed the product. In many industries it is accepted practice for customers - often potential customers - to audit suppliers, either against a quality standard or to gain confidence in particular processes. It would not be unreasonable for potential purchasers to audit suppliers of OTS products, not only against quality and safety standards, but also to assess their risk-analysis, business, and other processes. Indeed, there is a need for auditing to become an accepted custom, particularly in the software purchasing community, as numerous companies that provide software-based products have little concept of software engineering, apply minimal management to the process, and rely entirely on programmers of execrable standard (POES).

There is also the need for evidence to support claims that a supplier makes about software. For example, claims are now being made that software is of a given SIL. But there are many possible interpretations of such claims [10], so there is a need for a convention that embraces both definition and evidence.

Products may be claimed to be 'proven in use'. But such claims should not merely be stated; an argument and supporting evidence should also be provided. For instance, which functions are claimed to have been proven in use? What was the extent of the coverage of the use? What observational evidence is relied on for the claim? Which functions in the software have not been used or are not claimed to be proven? Are these functions independent of those for which the claim is made? Were the same version and configuration retained, without change, throughout the period of use, and are they the ones being offered now? We need a convention for the content and form
of proven-in-use arguments, and it might also include some design information, such as which functions are partitioned from others (so as to be independent of them) and how the partitioning is achieved.

A possible difficulty in applying a convention is that the safety-critical systems community does not wield significant influence on suppliers in the commercial market. But this is not an adequate reason for doing nothing. It is likely that many suppliers would see benefits in providing evidence about their products, as is currently the case in the supply of real-time operating systems. Where such suppliers lead, others are likely to follow if asked to do so.

7 Discussion

The COTS debate is about the suitability of OTS products for safety-related applications. At its heart is the lack of evidence to satisfy safety assessors of the appropriateness of OTS products, and an equally important issue is the nature of the required evidence.

Evidence of safety should be related to the risks involved. If the risk is small, the issue may be more of functionality than safety. Otherwise there must be evidence on which to base a safety case that satisfies the assessors. The standards call for development-process evidence, but COTS proponents argue that this is neither conclusive nor necessary, and that product evidence, adduced by testing or experience of use, can be as strong. In the absence of evidence to justify the use of an OTS item, a protection function may guard against its inadequacies.

COTS proponents claim advantages other than cost. They point out that OTS products are readily available and that the commercial market generates technological advances that are rapidly introduced into products. But if an OTS item is subject to frequent upgrades by its supplier, the cost of making and testing necessary changes and re-assessing the system could outweigh its benefits. Similarly, there are other disadvantages, throughout the life cycle, that could counter-balance the point-of-sale benefits of OTS products.

A frequently unarticulated side of the COTS debate is the argument against modern safety standards: if COTS products are deemed suitable for safety-related applications, can the standards' rigorous requirements for development-process evidence be justified? Neither assessment of the development process nor evaluation of the product can, in the current state of the art, provide absolute proof of safety. Perhaps they never will, for safety is application-specific and even 'good' products can threaten safety in inappropriate circumstances.

Whatever turn the COTS debate may take, evidence will remain crucial to safety assessment. It is argued in this paper that, rather than accept the assumption of its absence, we should define, and begin to put in place, a convention on what evidence should be provided by OTS-product suppliers in order that arguments for safety may be developed. Such a convention should also cover the evidence required in support of claims made by suppliers about their software, for example that it meets a SIL or that it is proven in use.

At a time when technological innovations are forcing us to revise our ways of thinking, and when attitudes to the traditional ways of developing and using software are changing, the COTS debate raises issues that must be confronted. It is a necessary
and healthy debate that will affect not only the ways in which we develop our safety-related software and systems, but also the ways in which we use them and justify their use. The considerable effort being expended on investigating how the use of COTS products can be justified will mean that, even without the prospect of early resolution, the COTS debate is likely to have many fruitful side effects, both theoretical and practical.

**Acknowledgement**

Thanks to Jim Armstrong whose comments and ideas have been influential on this paper.

**References**

An Investigation on Mutation Strategies for Fault Injection into RDD-100 Models

Mohamed Kaâniche¹, Yannick Le Guédart¹, Jean Arlat¹, and Thierry Boyer²

¹LAAS-CNRS
7 av. du Colonel Roche, 31077 Toulouse Cedex 4 — France
{kaaniche, arlat}@laas.fr

²Technicatome
BP 34000, 13791 Aix-en-Provence Cedex 3 — France
tboyer@tecatom.fr

Abstract. This paper focuses on the development of a conceptual framework for integrating fault injection mechanisms into the RDD-100 tool to support the dependability analysis of computer systems early in the design process. The proposed framework combines functional and behavioral modeling, fault injection and simulation. Starting from the RDD-100 model built by the system designers, two techniques are discussed for the mutation of this model to analyze its behavior under faulty conditions: a) insertion of saboteurs into the model, and b) modification of existing component descriptions. Four types of fault models are distinguished and specific mechanisms to simulate the corresponding fault models are proposed for each mutation technique. An approach combining the advantages of both techniques is proposed and a prototype implementing this approach is briefly described.

1 Introduction

Designing cost-effective fault-tolerant computer architectures is today one of the main concerns for the developers of dependable systems in a large segment of industrial control applications. However, besides evaluations based on probabilistic modeling or FMECA, the consideration in the early phases of the development process of dependability issues encompassing detailed behavioral analysis is still hardly supported in practice in industry, with few exceptions such as [1, 9]. Accordingly, we are currently investigating a method to assist fault-tolerant systems designers by incorporating the explicit analysis of their behavior in the presence of faults, in the early phases of the development process. The aim is to support designers in making objective choices among different high-level architectural options and associated fault tolerance mechanisms. The proposed approach is based on the development of a functional and behavioral model of the system, and on the behavioral analysis of the model by means of simulation and fault injection.

Several related studies have addressed the issue of supporting the design of dependable systems by means of simulation ([2, 3, 5, 6]). Although these approaches are generally supported by efficient tools, these tools are not designed to be included...
in the design process. Indeed, for cost and efficiency reasons, system designers would like to use the same formalism and tool to carry out preliminary functional and behavioral analyses, and to assess the effect of faults on the model of the system. This is why we have tried to attack this issue from a different perspective, i.e., to study instead how several existing design tools can be enhanced to support such an early dependability validation analysis. In contrast to the work reported in [1], that dealt with formal techniques (in particular, SDL), we focus here on a more pragmatic approach aimed at elaborating on the modeling and simulation capabilities of system engineering tools used in the industrial world by system designers. Statemate [7] and RDD-100 are among the various commercial tools that are currently used in industry. Mainly based on the preliminary experiments we have been carrying out to assess the suitability of both tools to analyze the dependability of systems in the presence of faults [11], our work is now focused on RDD-100.

The remainder of the paper is organized as follows. Section 2 deals with simulation-based fault injection and outlines the main reasons that led us to focus on model mutation. Section 3 summarizes the lessons learnt from the preliminary experiments that we carried out on real-life case studies. In particular, these experiments highlighted the need to define a generic approach to support model mutation. The feasibility of such an approach based on the RDD-100 tool is discussed in Section 4. Two model mutation techniques are considered: the use of dedicated fault injection components called saboteurs, and code mutation. Specific mutation mechanisms are proposed to implement each technique and are analyzed with respect to the fault models to be simulated. A comparison of both techniques is also provided. Finally, Section 5 provides the conclusion to the paper.

2 Simulation-Based Fault Injection

Starting from the original model (called nominal model) built by the designers to carry out preliminary functional and behavioral analysis of the system under nominal conditions, two main approaches can be considered for analyzing fault effects on the system behavior (e.g., see [8]). The first one uses the simulator built-in commands to alter the behavior of the model during the simulation (i.e., without modifying the nominal model). The second one consists in: a) mutating the nominal model before the simulation by adding dedicated mechanisms for injecting faults and observing their effects, and b) simulating the mutated model to analyze the effects of faults. For both approaches, the analysis of fault effects is done by comparing the traces obtained from simulating the system behavior under nominal conditions and faulty conditions, respectively.

The applicability and efficiency of the first approach, with respect to the types of faults that can be injected and their temporal characteristics, strongly depend on the functionalities offered by the command language of the simulator. However, the second approach can take advantage of the full strength of the modeling formalism,
i.e., any fault model that can be expressed within the semantics of the modeling language can be implemented. Moreover, this approach is well suited for the definition of generic fault injection mechanisms that can be included automatically in any model and transparently to the user. For these reasons, we have focused our investigations on the second approach.

3 Lessons Learnt from Preliminary Experiments

Several tools are used in industry to support functional and behavioral analysis of computer systems based on modeling and simulation, e.g., Statemate and RDD-100. To study the suitability of these tools for analyzing system behavior in the presence of faults, we carried out several experiments on four real-life case studies [11]. The four target systems were all related to critical applications from distinct fields (nuclear propulsion command control, ground-based and on-board space systems). The systems architectures were designed to satisfy stringent safety and availability requirements. They include several redundant components using voting and reconfiguration fault tolerance mechanisms.

In these experiments, we used RDD-100 and Statemate to:

- model some critical aspects of each system (e.g., synchronization, reconfiguration management) based on the specification documents;
- inject faults into the models (corrupting data, simulating delays and omissions, etc.);
- analyze the impact of these faults using the simulation engines integrated in the tool.

These experiments confirmed that major benefits can be obtained from the analysis of systems behavior in the presence of faults, early in the design process. For example, a model of a real-time ground-based space subsystem was developed to analyze the impacts of temporal faults on the system behavior. This experiment allowed us to reveal an error detection latency problem that was related to the overlapping of two error detection mechanisms and an incorrect sequencing of recovery actions.

From the point of view of fault injection implementation, these experiments highlighted the need to define a set of generic mechanisms allowing the mutation of system models in a systematic way, rather than on an ad-hoc basis. Indeed, to be applicable to the analysis of complex real-life systems in an industrial context, the following requirements should be satisfied by the mutation mechanisms dedicated to injecting faults and observing their effects:

1) The mutated model should preserve the behavior and properties of the nominal model if faults are not injected during the simulation. This will ensure that the mutation mechanisms will not alter the behavior of the nominal model in the absence of faults. Also, the comparison of the simulation traces obtained from the mutated and the nominal models will be made easier, as this comparison can be focused on the instants when faults are injected.

2) Model mutation should be based on the semantics of the modeling formalism, and not on the target system model. The objective is to provide generic mutation
mechanisms that are applicable to a large set of systems, rather than to a particular system.

3) Model mutation should be performed automatically and transparently to the user. The latter should be involved only in the specification of the faults to be injected and the system properties to be analyzed.

The RDD-100 formalism was chosen by our industrial partner to support the definition of this approach. Our investigations concerned: 1) the definition of model mutation mechanisms based on the RDD-100 formalism, and 2) the development of a prototype tool that integrates those mechanisms into RDD-100. The rest of this paper summarizes the main results obtained from our study.

4 Fault Injection into RDD-100 Models

Two different techniques for model mutation have been investigated. The first one is based on the addition of dedicated fault injection components called “saboteurs” to the RDD-100 nominal model. The second one is based on the mutation of existing component descriptions in the RDD-100 nominal model. Before describing these two techniques, we provide first a summary of the main concepts of RDD-100 formalism.

4.1 RDD-100 Main Concepts

RDD-100 incorporates a coherent set of formalisms that enable engineers to: 1) define requirements and allocate them to system components according to a hierarchical approach, 2) refine their behavioral description into discrete processes and allocate these to system interfaces, 3) establish system feasibility on the basis of resources and costs, and 4) iterate the engineering design process with increasing levels of detail. Our study focuses on the behavior diagrams formalism (named F-Net) defined within RDD-100 to support the detailed description and simulation of system and components behavior.

A behavior diagram is a graphical representation of the modeled system, composed of a set of concurrent and communicating (SDL-like) extended finite-state machines, called Processes. The finite-state machine is extended in the sense that it has memory and can make decisions about responses to stimuli based upon that memory, and communicating in the sense that it can stimulate other processes by sending messages to them. The behavior of each process can be detailed by describing the set of Functional Modules executed by the process and their interactions. Functional modules whose behavior can be described hierarchically are called TimeFunctions and those that are not further decomposed are called DiscreteFunctions. Three types of items can be input to or output from a function: Global, State, Message. Global items are available to all functions within the model. State items are available only to the functions of the same process. Messages can be sent only from one process to another. Each function can receive at most one message. However, it can send several messages, one per recipient.
Figure 1 presents a simple example of a behavior diagram composed of two concurrent processes: Process 1, composed of module F, and Process 2, composed of module G. Concurrency is represented by the node (&). At the beginning of the model execution, a token is produced by the simulator. The token flow through the model structure describes different execution scenarios. When a token reaches the top node of a parallel branch, the simulator creates a new token for each process. The simulator treats each process concurrently. When the tokens of all processes reach the bottom node (&), the simulator recombines them into a single token and advances the token to the next node on the diagram.

Functional modules execution is initiated when the token and the input messages (if any) reach the module. Then, the data transformations, defined in the functional part of the module (coded in SmallTalk) are processed, i.e., reading and writing data in the state variables and sending messages (if any) to the subsequent output modules. The outputs become available once the module execution time defined in the module code is elapsed.

![Fig. 1. Example of a behavior diagram](image)

RDD-100 provides three constructs or structures (Select, Iterate, and Replicate) allowing for a concise representation of complex behavior diagrams. The Select structure enables to choose among multiple execution paths at the output of a given module depending on conditions specified in the code of that module. The Iterate structure describes the repetitive execution of the sequence of modules that appear between two iteration endpoints. Finally, the Replicate structure allows the representation of equivalent processes by a single abstraction in the model. This structure is particularly useful for the modeling of redundancy in fault tolerant systems.

4.2 Fault Models

Fault injection requires the definition of faults types, fault activation time and fault duration. Different types of faults can be injected to alter the value or timing characteristics of behavior diagrams. In our study, four types of faults are distinguished:
• data corruption of global items, state items, or messages, that are input to or output from a functional module;
• delayed execution of a functional module;
• non activation of a module when triggered;
• spurious activation of a module; i.e., the module is activated whereas it is not supposed to.

For each fault type, its activation time can be related to the simulation time or to the model state and its duration may be permanent or transient.

4.3 Model Mutation Based on Saboteurs

In this approach, each functional module of the RDD-100 nominal model is considered as a black box. Fault injection is carried out by dedicated components associated to the target module, called saboteurs. The saboteurs intercept the inputs or the outputs of the target module and possibly alter their value or timing characteristics to imitate the behavior of the module in the presence of faults.

In the following, we describe the approach that we defined to mutate RDD-100 models based on the insertion of saboteurs. This approach aims at satisfying the requirements listed in Section 3. We first illustrate the mutation of a single functional module, then we discuss how this approach can be applied when the nominal model includes special constructs such as Replicate and Select.

4.3.1 Mutation of a Single Functional Module. For each functional module of the nominal model, we associate two saboteurs: S1, intercepts the inputs of the module and possibly alters their value or timing characteristics, and S2 acts in a similar manner on the outputs of the module. The types of faults to be injected in the target module, as well as their activation time and duration are specified in the functional code of the saboteurs. As illustrated in Figure 2, the mutation consists in transforming the target module F, into three parallel processes, corresponding to the execution of S1, F and S2, respectively. The communication between the saboteurs and F is done through message passing. This mechanism enables the synchronization of the functional module with its associated saboteurs. Indeed, module F execution is initiated when S1 terminates, and S2 is activated after the execution of F. Therefore, the input items sent to F (i.e., messages, Global and State items) may be altered by S1 according to the fault model specified in S1, before execution of F. The analysis of the outputs delivered by the module will allow the designers to analyze the impact of the simulated faults on the module behavior as well as on the system. Similarly, faults can be injected on the module outputs to assess to what extent such faults can be tolerated by the system. If the fault activation time is conditioned upon some Global or State items that are not included in the inputs to module F, an additional input to S1 and S2 is added to include these items as illustrated on Figure 2-b. All this process can be performed automatically.

When it is not activated, the behavior of each saboteur is transparent, as if it was not present in the model. The input data are delivered instantaneously to the functional module through S1 without any modification. Similarly, the outputs of the module are
made accessible to the model components, through S2. Therefore, the saboteurs remain inactive until a fault is triggered.

![Diagram of Module F before saboteurs insertion](image1)

![Diagram of Mutated Module](image2)

**Fig. 2.** Mutation of a single module

4.3.2 **Injecting the Four Types of Faults.** The injection of the four types of faults defined in § 4.2 is performed as follows.

Data corruption can be achieved by substituting new values to the original ones. Any data item or data type defined within a behavior diagram can be modified accordingly.

A delayed execution of a module is simulated simply by assigning to the saboteur an execution time corresponding to the value of the delay.

The simulation of the non execution of a module when triggered is done by means of S2. S2 intercepts all the module outputs and ensures that the module remains silent during the duration of the fault.

Finally, the simulation of the spurious activation of a module is more problematic. Indeed, to be activated, a module must receive the activation token as well as the input messages expected by the module (if any). Two solutions can be considered to simulate such a behavior. The first one consists in modifying the behavior diagram to ensure that from each node of the behavior diagram there is a link leading to the target module over which the activation token can flow to activate the target module upon the occurrence of the fault. This solution is not practically feasible, especially when we deal with complex behavior diagrams. Also, it requires a significant modification of the internal code of the functional modules to ensure that the resulting model is consistent. For the second solution, we assume that only the modules of the nominal model that receive input messages may exhibit such a behavior. In this case, the spurious activation of the module can be simulated by means of a specific saboteur, designed to send a message to the target module(s) upon the activation of the fault. This solution is illustrated in Figure 3.

In this example, the nominal model is composed of two processes corresponding to the activation of modules F and G respectively. To simulate a spurious activation of G at some time t, we create a saboteur that is executed concurrently to modules F and G, and remains active during the simulation. This is ensured by using the iterate structure represented by the `@` symbol. When the fault activation time is reached, the saboteur...
sends a message to the target module (G). This message will be taken into account by G when the token is received.

So far, we did not discuss how to mutate the model when some input data are shared by several modules. In this case, different situations can be distinguished; for instance:

1) The fault affects the input interface of one module only.
2) The fault affects the input interfaces all the modules, and the same error is produced.
3) The fault affects the input interfaces of all the modules, but different error symptoms are produced at each interface; this might correspond for example to the occurrence of byzantine faults.

The mutation mechanism proposed in Figure 2 reproduces the faulty behavior corresponding to the first situation. Indeed, the saboteurs associated to a given module are designed to alter the execution context of the target module without altering the execution context of the other components of the model. Therefore, if we corrupt an input data that is shared by the target module and other components, the modified input will be propagated to the target module only. The other components will still perceive the original copy of this input. This choice offers more flexibility to the users with respect to the kind of faulty behavior they would like to simulate. In particular, the faulty behaviors corresponding to situations 2) and 3) described above, can be easily simulated by associating saboteurs to each component, and specifying the attributes of the faults to be injected in the input saboteur according to the faulty behavior to be simulated (i.e., same error patterns for situation 2 and different error patterns for situation 3).

4.3.3 Modules without Input or Output Messages. In § 4.3.1, we assumed that the functional module to be mutated has input as well as output messages. However, the nominal model might include some modules that do not have messages either in the input or in the output domain. In this case, we just need to create dummy messages, between S1 and the module, or between the module and S2. Once the messages are created, the mutation is performed according to the approach described in § 4.3.1. It is noteworthy that the content of the dummy messages is irrelevant, as the
only role of these messages is to synchronize the execution of the module and its associated saboteurs.

4.3.4 Functional Modules with Multiple Output Messages. An RDD-100 module can receive at most one input message. However, it can send multiple output messages, one per recipient. To mutate a module with multiple output messages, we have to adapt the construction proposed in Figure 2 to ensure that no more than one message is sent to the output saboteur, S2. Two techniques can be considered to satisfy this condition. The first one consists in modifying the internal code of the target module to ensure that only one output message is sent to S2. This can be achieved by transforming all the output messages, but one, into Global items. The remaining message will be used to trigger the saboteur S2. When the Global items are accessed by S2, they are transformed into messages before being delivered to their recipients. The second technique consists in associating a saboteur to each output message, and coordinating the execution of these saboteurs. If the number of output messages to be altered is important, this could lead to very complex mutated models. Clearly, the first solution is more practical, even though its implementation requires the modification of the functional code of the target modules.

4.3.5 Mutation of a Replicate Structure. The Replicate structure is a notation used to specify the simultaneous simulation of multiple copies of the same process within a behavior diagram. It is composed of three parts:
1) A replicate branch defining a sequence of functions that represents a process. This process is treated by the simulator as having multiple copies (replicates) that are simulated concurrently.
2) A domain set defining the number of replicates created when the token enters the replicate structure.
3) A coordination branch controlling the execution of the replicated processes. The coordination branch may send messages to and receive messages from the replicated processes, and may create and delete replicate processes. Any message received by a replicate process must be passed through the coordinate branch.

Two targets can be considered for the mutation of a Replicate structure: the replicated modules and the coordination module. The mutation of each of these targets is performed according to the process presented in previous Subsections, i.e., two saboteurs S1 and S2 are associated to each mutated module. The mutation of a Replicate structure leads to the replication of the saboteurs associated to each module of the structure. In this context the same faults will be injected in each replica. However, it is also possible to access distinctly each mutated replica, and specify different faults for each mutated module.

The mutation of the coordination module offers to the users several possibilities to alter the global behavior of the replicate structure. Besides altering the content and the timing characteristics of the messages exchanged by the replicas, the mutated coordination module can be used to dynamically remove some replicas from the execution process (e.g., because of failures) or to increase the number of active replicate (e.g., as a result of recovery actions). Thus, this mechanism is particularly useful to analyze the behavior of fault tolerant systems under faulty conditions.
4.3.6 Mutation of a Select Structure. The Select structure is represented by the notation (+) and allows for performing selectively either one process or the other, depending on the arrival sequence of messages from other processes. For example, in Figure 4-a, after the execution of module F, the token may be directed to the branch executing F1 or to the branch executing F2 depending on conditions specified in the F user code.

To support fault injection, each module F, F1 and F2, can be mutated according to the process described in Section 4.3.1. For example, Figure 4-b presents the model obtained when only F is mutated. With this model, any modification of module F inputs remains local to that module and does not affect the inputs of F1 or F2. If one wants to reproduce the same perturbation on the inputs of F1 or F2, two alternatives can be considered:

1) Specify the same kind of fault to saboteurs associated to F1 or F2.
2) Modify the mutated model in Figure 4-b to ensure that any modification of an input item shared by F and the modules involved in the select structure (F1, F2) affects all these modules. This can be done by directing the outputs of saboteur S1 associated with F, to the input interfaces of modules F1 and F2.

![Fig. 4. Mutation of a Select structure](image)

The non activation of a module when triggered can be simulated by ensuring that the outputs of the module are not modified when the fault occurs. If we inject such a fault into module F in Figure 4, its outputs will not be updated when the fault occurs and the token will be directed to either F1 or F2 depending on the system state. Therefore, injecting such a fault does not mean that the whole Select structure is not executed when the fault occurs. If one wants to simulate the latter behavior, it is necessary to modify the internal code of F, i.e., such behavior cannot be simulated simply using saboteurs.

4.4 Mutation of the Code of the RDD-100 Modules

So far, we assumed that each module of the RDD-100 nominal model was a black box and we analyzed how to mutate the model by inserting saboteurs dedicated to injecting
faults at the input or output interfaces of the target modules. In this section we analyze how to mutate the internal code of the nominal model by including the code dedicated to fault injection without inserting additional components into the model.

Code mutation offers a variety of possibilities to alter the behavior of the nominal model. Besides altering the data values and timing characteristics of the target modules, any statement of the original code can be corrupted, e.g., substituting operators or variable identifiers; this is similar to the mutation techniques used by the software testing community [4, 12, 13]. In this paper, we focus only on the fault models defined in § 4.2.

**Data corruption.** Any input or output data item (global, state or message) manipulated by a module can be easily accessed. Data corruption consists in substituting to the target items new values corresponding to the faults to be injected. A simple example is presented in Figure 5. The user code of the target module consists in reading two input data items A and B, computing their sum and writing the result C to the output interface (Figure 5a). The mutated code leading to the corruption of A is given in Figure 5b. The output value C can also be altered using the same process. More generally, to reproduce the same faulty behavior implemented with the saboteurs, we have to ensure that when we modify the value of a data item, any occurrence of this data item in the original code is substituted by the new value. This process can be easily automated. Clearly, this technique is easier to implement and is more efficient than the approach based on the insertion of saboteurs discussed in § 4.3.

![Fig. 5. Data corruption by code mutation](image)

**Delays.** The simulation of delays can be implemented in two ways: 1) either the execution of the module is delayed by inserting a delay at the beginning of the code, i.e., the input items are read by the target module when the delay specified in the mutated code expires, or 2) the emission of output items to the output interface is delayed. To implement the latter case, we have to identify in the original code each output statement and insert a delay before the corresponding outputs can be accessed.

**Nonactivation of a module when triggered.** This kind of behavior can be simulated easily by identifying and deactivating all or selected output statements in the target code.
Spurious activation of a module. The simulation of this behavior requires the identification of the modules that send messages to the target module, and then mutate the corresponding code by inserting a statement that forces the emission of a message to the target module when the activation time of the fault is reached. When more than one module is able to activate the target module, an algorithm must be implemented to decide, which module will be selected to send the triggering message when the fault is activated. Clearly, the implementation of this strategy is complex. The solution presented in § 4.3.1 that is based on the definition of a saboteur dedicated to the activation of the target module appears thus more suitable to simulate this kind of behavior.

Mutation of Replicate and Select structures. The code mutation mechanisms discussed above for a single functional module can also be applied to mutate Replicate and Select structures, with the same advantages and limitations. More details are provided in [10].

4.5 Comparison and Discussion

In this section, we analyze to what extent the two mutation strategies presented in § 4.3 and § 4.4 satisfy the guidelines defined in Section 3.

4.5.1 Preservation of the Nominal Model Properties. The mutation mechanisms based on the saboteurs or on the modification of the RDD-100 components code are designed to be inactive when faults are not injected during the simulation. Considering the first mutation approach, the saboteurs are executed instantaneously and the input data as well as the output data are delivered to their recipients without being altered. If we except the additional traces resulting from the modification of the behavior diagram structure due to the insertion of the saboteurs, the outputs provided by the simulation of the mutated model and the nominal model will be identical (in the value and timing domain). Therefore, the properties of the nominal model are preserved by the mutated model when it is simulated without activating faults. These properties are also preserved by the mutated model obtained with code components mutation. Note that no additional traces are produced with the latter approach due to the fact that the model structure is preserved.

4.5.2 Independence with Respect to the Target Systems. All the mutation mechanisms described in the previous sections were defined based on the characteristics of the RDD-100 behavior diagrams formalism and the SmallTalk language used to implement the code of RDD-100 components. They are generic in the sense that they can be applied to any system model built with the RDD-100 tool. Nevertheless, the specification of the types of faults to be injected and of the properties to be analyzed naturally requires a deep knowledge of the target system.

4.5.3 Transparency. Starting from the nominal model, and the specification of the fault injection campaign to be performed, the generation of the mutated model based on the insertion of saboteurs or on the modification of components code can be carried
out automatically. A set of mechanisms integrated within the RDD-100 tool can be developed to support the implementation of the mutation strategy. A prototype tool is currently under development to implement this approach [10].

4.5.4 Comparison of the Proposed Solutions. The two mutation techniques discussed in the previous sections present some advantages and limitations.

Considering saboteurs, the code dedicated to fault injection is separated from the original code. This facilitates the analysis of the mutated model. In particular, the graphical representation of the mutated model clearly identifies the components to be altered as well as their characteristics. Moreover, the saboteurs can be defined as reusable components. Thus, the implementation of model mutation algorithms can be simplified significantly. However, a major problem with this technique is related to the dramatic increase of the complexity of the mutated model due to the insertion of additional components. As regards the fault modeling capacity of this technique, we have identified some situations that cannot be easily handled simply by using saboteurs and without modifying the original code of the target modules (e.g., functional module with multiple output messages). This is related to the fact that the saboteurs have a restricted view of the target components, i.e., fault injection can be performed only through the input or the output interfaces of these components.

The code mutation technique does not have such limitation. Indeed, any fault model can be simulated, provided it can be expressed within the RDD-100 formalism. Generally, for simple fault models we only need to add a few statements to the target code to support fault injection. Conversely, there are some situations where it is more suitable to use saboteurs to describe the faulty behavior to be simulated. This is the case for example of the simulation of the spurious activation of a target module (See § 4.4).

Clearly, the above discussion shows that the combined use of saboteurs and code mutation provides a more comprehensive and flexible approach for mutating RDD-100 models. A prototype tool implementing such an approach is currently under development. The prototype is implemented in Perl and performs the following tasks:

1) Analysis of the nominal model
2) Set up of fault injection campaign
3) Generation of the mutated model

Tasks 1 and 3 are performed automatically without any interaction with the user.

The analysis task consists in parsing the textual description of the nominal model and identifying all the model components (processes, TimeFunctions and DiscreteFunctions), the inputs and outputs associated to each component (State items, Global items, Messages) and the interactions between these components (data flow and control flow). The outcome of this analysis is a list of potential targets for fault injection. Based on this list, the user can select the components on which fault injection will be carried out and specify the kind of faults to be injected with their activation time and duration. Based on this specification, the mutated model is automatically generated by the prototype. Code mutation is used to simulate data corruptions, delays, and non activation of modules when triggered. The simulation of an inopportune activation of a module is carried out by using a dedicated saboteur. More details about this prototype are given in [10]
5 Conclusion

The main objective of our study consists in integrating fault injection mechanisms into system engineering tools actually used in industry by system designers. By doing so, dependability analyses can be fully integrated early in the development process of dependable systems.

This paper focused on the development of a systematic method for integrating simulation-based fault injection mechanisms into RDD-100 models to support system designers in computer systems dependability analysis. Starting from the functional and behavioral model developed by the designers, the proposed approach consists in mutating the model by including mechanisms aimed at injecting faults, and simulating the mutated model to analyze the impact of injected faults on the system behavior.

Two mutation techniques have been studied: the first one consists in adding fault injection components called saboteurs, that are designed to alter the input or output interfaces of the target components, and the second is based on the code mutation of original model components. Four types of fault models have been considered (data corruption, delay, non activation of a module when triggered, and spurious activation of a module) and specific mechanisms have been proposed for each mutation technique to simulate the corresponding fault models. The comparative analysis of techniques showed that a more practical and flexible approach should combine both techniques, instead of using either one or the other. In particular, code mutation can be used to simulate data corruptions, delays, and non activation of components when triggered, while saboteurs are more suitable to simulate a spurious activation of some target components. A prototype implementing this approach is currently under development.

Acknowledgments. Yannick Le Guédart was supported by Technicatome. This work was carried out at the Laboratory for Dependability Engineering (LIS). Located at LAAS, LIS was a cooperative laboratory gathering five industrial companies (Astrium-France, EADS Airbus, Électricité de France, Technicatome, and THALES) and LAAS-CNRS.

References


A Comparison Study of the Behavior of Equivalent Algorithms in Fault Injection Experiments in Parallel Superscalar Architectures

Ioannis Vakalis

EC-Joint Research Centre
Institute for Systems, Informatics & Safety, TP 210 21020 Ispra (Va), Italy
ioannis.vakalis@jrc.it

Abstract. This work describes an experimental comparison study of the behaviour of a set of algorithms in the presence of faults. The algorithms have the characteristic that they belong to the same problem class and a number of comparison studies exist in bibliography, with respect to their numerical, conversion and time and space complexity. The class of the used algorithms solve the matrix exponentiation problem. This is a well studied numerical problem encountered in the area of linear differential equation, with a number of solving algorithms. For this study we use Fault Injection techniques at the compile time, namely software based script fault injection method based on random bit inversion assumption. The experiments are performed on a fully operating, parallel architecture machine with two superscalar processors. The algorithms are studied for their fault tolerance in the presence of an elementary fault detection method based on command duplication and exploiting the parallel architecture.

Keywords: Fault injection, equivalent algorithms, parallel superscalar architecture.

1 Introduction

The increasing presence of computing equipment in numerous domains of daily life calls for a reduced rate of failure and pushes the dependability standards higher. The demand for low-cost reliable computing devices embedded in a large spectrum of apparatus makes the question of operating costs (including damage costs) imperative.

Software is known to be one of the main causes of problems in the reliability of a computing device [1][9].

The complexity of the size of present software systems raises severe obstacles to the analysis of failure scenarios. To overcome this problem in the recent years researchers have relied on experimental techniques to assess the dependability of computing systems [4]. A low cost technique showing promising results used frequently lately is that of Fault Injection.

There are mainly two categories of methods of Injecting Faults into a system, software based [3][6] and hardware based [4][5][7]. Each of them has a number of variations. Hybrid methods have also been developed [13][18].

One interesting aspect to look at is the difference in the performance in fault injection of algorithms intended to solve the same problem. Comparing the results of
algorithms belonging to the same group in fault injection experiments can reveal interesting results and can offer a new insight in algorithm design methods.

This type of investigation can illuminate aspects of the algorithms that make them either more fault tolerant or instead more fault prone. A thorough examination of the characteristics of each algorithm could give new clues for fault tolerant algorithm design.

This paper investigates the performance of a set of algorithms solving the same problem. Particularly software based fault injection techniques are applied on a number of similar algorithms, which solve the same numerical problem. There exist a number of computational problems that can be solved with diverse algorithmic approaches. This is the result from different types of analysis done on the problem in an effort to resolve certain computational and mathematical issues like convergence, time and space complexity and stability.

Matrix computations are a very important part of the numerical computations area with a vast number of applications in different scientific fields. Many works have been done for matrix algorithms in terms of fault tolerance [11]. In the fault injection area, nearly every experimental study includes a matrix computation algorithm since they are considered good benchmarks. For the same reasons we have chosen the algorithms for our experiments from the matrix computations field.

In this work a number of algorithms has been selected, representing a solution to the same problem namely the matrix exponentiation \(e^A\). This is a very common computational problem and has produced a vast literature in the past as well as a large number of algorithmic solutions (19 thus far [15]). This computation is encountered quite often in control systems and in real-time systems where a time varying version of the matrix exponential is computed \(e^{At}\).

Our intention is to investigate the behaviour of these algorithms in random faults and in addition their performance under fault simple fault tolerance scheme. For this purpose we are exploiting the parallel architecture and the parallel techniques of the experimental set-up.

2 Fault Injection Process Description

Since our target is to study the behaviour of certain algorithms in the presence of faults, the technique of software fault injection seems more appropriate. The basic experimental set-up includes a procedure that allows code corruption at the software level. This fact is important because our target is to study the effect of the different structural features of the algorithms. In addition this choice will lessen the shortcomings of the software fault injection methods discussed in earlier works [1]. The experimental setting is an otherwise fully operational UNIX based system with two parallel processors used mainly for scientific calculations.

The experiments belong to the category of executable code corruption, software based, one bit–flip error injection.

We injected faults using a shell script file that according to the result of a random number generator alters one bit chosen randomly in a byte, also chosen randomly. The executable is “edited” so that the selected bit is altered and then is executed to observe the consequences. This is a “compile-time” type of software fault injection and does
not allow us to study the results of faults during workload run-time [1]. We will address this issue in a future work.

A basic characteristic of our fault injection method is that in this stage we do not differentiate between errors that happen in data and in commands. We face the whole of the code in a uniform manner.

**Fig. 1.** Fault injection scheme process flow

A basic characteristic of our fault injection method is that in this stage we do not differentiate between errors that happen in data and in commands. We face the whole of the code in a uniform manner.
3 Fault Injection Environment

The Fault Injection environment, we used in our experiments is built around a UNIX based system namely a SGI Octane® workstation. This system includes 2 CPUs at 250 MHz MIPS R10000 main memory of 512MB, Data Cache 32 KB, and Instruction Cache 32 KB. The operating System is IRIX®64 Rel 6.5. The two microprocessors are MIPS® R10000 is a 4-way superscalar architecture. This type of architecture offers a number of special advantages as described in [14][21][22].

The algorithms we use in our experiments are written in C and the compiler is used with the optimization flags off to avoid any alteration in the main structure of the algorithms, which could affect the outcomes of the experiments.

Another possibility is to work at the assembled code level, which allows a better control over the system both of the fault injection process as well as the error detection technique [19]. This type of experiments will appear in another publication.

The system operates disconnected from any network node. In addition all user and administrator activated processes are stopped in order to avoid conflicting errors.

4 The Algorithms

Modelling of many processes with a system of ordinary differential equations involves the equation:

\[ \dot{x} = Ax(t) \]

where \( A \) is an \( n \times n \) matrix and the solution to this problem is:

\[ x(t) = e^{At}x_o \]

where \( e^{At} \) is the convergent power series:

\[ e^{At} = I + At + \frac{A^2t^2}{2!} + \cdots \]

There have been developed many algorithms to compute this quantity based on results from classical analysis, matrix theory and approximation theory. The three algorithms we have chosen are representative of the main groups of solving methodologies [15]. The different methodologies have been developed in an effort to tackle various issues like generality, reliability, stability, accuracy, efficiency and simplicity.

The selected algorithms for our experiments are the following:

• **Taylor Series**: This algorithm belongs to the group of Series Methods. It is a brute force algorithm to calculate converging series that demonstrates numerical problems and not so good stability [15].

• **Inverse Laplace Transforms**: The second algorithm belongs to the group of Polynomial Methods. It is based on the computation of the Laplace transform of the matrix exponential. It is considered as more effective algorithm but still has some of the defects of the previous group since in the second part it
computes series In the first part computes the matrix components using the Leverrier-Fadeeva algorithm [15].

- **Schur Decomposition**: The third algorithm belongs to the group of *Matrix Decomposition Methods*. This group is considered the most effective since it does not rely on series convergence. The main problem in computation is when the matrix has repeated eigenvalues or eigenvalues close to each other [20]. It uses matrix transformation method to transform the matrix to upper triangular form (Schur). And in the final step uses the matrix exponential of the upper triangular matrix and the transformation matrix. The matrix exponential of the upper triangular form can be computed using the Parlett algorithm. This is considered one of the most stable and numerically effective methods [15].

In summary the first algorithm is a series convergence algorithm and the third one is a decomposition algorithm. The second one lies between the two being partially series partially decomposition algorithm.

For reasons of simplicity and without loss of generality in this work we keep $t=1$. The structure of the algorithms can be better described with the following pseudocode Fig. 2.

```c
void mat_exp_power_series()
{
    initialize
    expA=I+A;
    for(i=0; i<n; i++)
    {
        while(conv_error > too_small)
        {
            new_term=A^k / k!;
            expA = expA + new_term;
            conv_error=test(new_term);
        }
    }
}

void mat_exp_laplace_series()
{
    /* Compute $B_0, \ldots, B_{n-1}$ */
    for(i=0; i<n; i++)
    {
        $B_i=$Leverrier_Fadeeva($A, B_{i-1}$);
    }
    for(i=0; i<n; i++)
    {
        while(conv_error > too_small)
        {
            $c_{ik}=\text{new\_term}(c_{ik-1}, \ldots, c_{ik-n-1})$;
            new_term_poly_i = $c_{ik}/k!$;
            $poly_i += new\_term\_poly_i$;
            conv_error=test(new_term_poly_i);
        }
        expA=expA+poly_i * $B_i$;
    }
}
```
The three tested algorithms in pseudocode

```c
void mat_exp_schur_decompose()
{
    initialise
    /* Schur decomposition $A=Q S Q^T$ */
    \{Q,S\}=schur_decomp(A)
    expS=Parlett(S);
    expA=Q*expS*Q^T;
}
```

**Fig. 2.** The three tested algorithms in pseudocode

## 5 Experimental Results

In the following we describe the experiments, used in order to evaluate the structural characteristics of the algorithms in the presence of faults.

We injected 2000 errors and the matrix that was used as an example for the computations was a $10 \times 10$ matrix generated randomly. This matrix was chosen among a set of matrices because it had distinct eigenvalues [20] and it did behave numerically stably in the computations.

The effect of injected faults on the behavior of the algorithms when processing numerically unstable matrices will be investigated in the near future.

All the compiler optimizations have disabled when we compiled the code for all three algorithms.

The results for the first set of experiments have been grouped and tabulated in order to compare the behavior of the algorithms.

We have classified the results of the experiments in the following categories:

- **Correct Results:** the code terminates normally and produces the correct results.
- **Wrong Results:** the code terminates normally but produces wrong results
- **System Hangs:** the system either gets stuck or enters into an infinite loop
- **Core Dump:** the system terminates abnormally mainly because, either of bus error or memory error. Rarely there are core dumps because of an illegal instruction.

The category of Correct Result is also known as *Fail-Silent* faults [13]. Table 1. includes the results showing that a quite large percentage of the injected errors does not produce wrong results (more than 62% for all the algorithms). This can be explained by the fact that the bit inversions affect large parts of code that are not important for the execution of the code (e.g. the name of variable). This is in accordance with the fact that the *Taylor Series* algorithm demonstrates the largest percentage of correct results. It is the simplest algorithm and therefore has the least number of “crucial” points in the code.
Table 1. Results of the first set of experiments

<table>
<thead>
<tr>
<th></th>
<th>Correct Results</th>
<th>Wrong Results</th>
<th>Core Dump</th>
<th>System Hang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taylor Series</td>
<td>1361</td>
<td>104</td>
<td>523</td>
<td>12</td>
</tr>
<tr>
<td>Laplace Series</td>
<td>1245</td>
<td>97</td>
<td>650</td>
<td>8</td>
</tr>
<tr>
<td>Schur Decomp</td>
<td>1344</td>
<td>153</td>
<td>494</td>
<td>9</td>
</tr>
</tbody>
</table>

Fig. 2. The comparative results expressed in % of three tested algorithms

Another characteristic of the results is the high percentage of core dump for the Leverrier-Fadeeva method. This can be a direct consequence from the fact that there exist a greater number of large do-loops (two of them are nested) compared to the other two algorithms.

The Schur Decomposition algorithm demonstrates an increased percentage of wrong results. This can be explained by the fact that the basic characteristic of the Schur method is that it includes a large number of matrix pivoting operations. Where, each of the pivots depends on the numerical results of the previous. This could be a hint that an effort to gain numerically stability leads to vulnerability to data errors. This issue requires a further investigation.

Finally the hung results are almost equal for all the algorithms. In Fig. 2 we can see the bar diagram indicating the results in percentage showing better the similarities and differences of the discussed results.
6 Exploiting the Parallel Architecture for Fault Tolerance

The computational environment we described in section 3 offers the possibility to evaluate the same set of algorithms by including an elementary fault tolerance mechanism. Following the simple recipe of code duplication we evaluated the behavior of the same set of algorithms. The detection mechanism merely compares the results of the two copy portions of the code after their execution. The performed fault injection experiments allowed us to observe some also interesting results and to compare them with the results of the first set of experiments. In this part of the testing we repeated 2000 fault injection experiments with the same random method we described in section 2.

For the Fault Tolerance mechanism we exploited also the parallel architecture of the system and the parallel programming facilities, which are offered by the MIPS C compiler.

We actually duplicated all the part of the code that does the computation of the matrix exponential in the programs. Then using the `#pragma parallel` programming option of the compiler we forced the two portions of the code to execute each in a separate processor, but in different thread each time. This way we used code duplication and at the same time we isolated the two parts of the code as possible. To execute in one processor we used the `#pragma one processor` structure of the compiler. Parallel and concurrent methods of Fault Tolerance have been investigated extensively in recent years [12].

The basic side effect of this construction was that the size of the code almost doubled and the execution time slowed down considerably.

The categories of the errors change slightly since we need to incorporate the Fault Tolerance error detection mechanism. More precisely the Wrong Results have to be replaced by two other categories the Caught Error and Uncaught Error. Those correspond to the situation that the Fault Tolerance mechanism can or cannot detect the error result. The second category is more known as Fail-Silent Violations [10][13].

Table 2. The results of the experiments with the Fault Tolerance Parallel Code added on the three algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Correct Results</th>
<th>Uncaught Wrong</th>
<th>Caught Wrong</th>
<th>Core Dump</th>
<th>System Hang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taylor Series</td>
<td>1327</td>
<td>56</td>
<td>17</td>
<td>590</td>
<td>11</td>
</tr>
<tr>
<td>Laplace Series</td>
<td>1129</td>
<td>76</td>
<td>53</td>
<td>723</td>
<td>20</td>
</tr>
<tr>
<td>Schur Decomp</td>
<td>1302</td>
<td>121</td>
<td>26</td>
<td>541</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2 summarizes the results. What is interesting to see is that the correct results have reduced in all algorithms and instead the core dump results have increased. This can be explained by the fact that the errors affecting bus and memory access have increased which are actually the parts of the code that make the computations thus more sensitive to errors. Also in the Laplace Series algorithm the core dump
increased more than the others leading to the conclusion that this algorithm has more intense bus and memory access than the others.

As expected the code duplication reduced the percentage of the wrong results but lead to undetected error percentage for more than 60% for all the algorithms (77% Taylor, 60% Leverrier, 82% Schur). Which is not an encouraging result.

**Fig. 3.** The results in percentage with Fault Tolerance for the three tested algorithms

Finally system hangs doubled for the Laplace Series algorithm instead it remained the same for the other two algorithms.

To summarize, the experiments using the fault tolerance mechanism based on code duplication and parallel processing showed that the structural characteristics of the three algorithms also affect the results of the experiments [19]. The results of Laverrier-Fadeeva algorithm can be considered as interesting in the sense that we do not see a proportional change in percentages as with the other two algorithms (Fig. 3).

7 **Conclusions and Future Work**

We planned and executed a series of Fault injection experiments targeting an approach for automatically transforming programs written in any high-level language so that they can be able to detect most of the errors affecting data and code.

We were able to draw some conclusions with respect to the structure of the algorithms but a more detailed analysis is required at instruction level. The second set of the experiments showed that the code duplication with parallel execution does not improve substantially the fault tolerance of the three algorithms. The difference in the level of improvement among the three algorithms indicates another direction for future research.
A clearer picture can be drawn about the results if we analyze further and we seek more detailed information on the causes of the observed behavior of the algorithms. This is possible if we consider different categories of errors, which affect certain categories of statements particularly at the machine instruction level. In this line of thought belongs the following categorization of statements and errors [16]. According to this the statements can be divided in two types:

- Statements, affecting data (e.g., assignments, computations, etc.)
- Statements, affecting the execution flow (e.g., tests, loops, function calls etc.).

The errors affecting the code can be divided in two types, depending on the way they alter the statement:

- Errors, changing the instruction to be executed by the statement, without changing the execution flow (e.g., an add operation into a sub)
- Errors, changing the execution flow (e.g., an add operation into a jump).

This classification is presented in [16][17]. A classification of this type is necessary to categorize the types of errors and their effect. It is possible to create fault lists that allow the injection of more specifically targeted errors [8]. Possibly a more refined classification is required for a thorough analysis.

Our intention for future research is to study also the effects of injected errors on the performance applications running on a R10000 MIPS and other similar architecture processors. This group of processors offers hardware support for counting various types of events (cache misses, branch mispredictions etc.). These events are directly related to the performance measure of an application [13].

8 Acknowledgements and Disclaimer

The author would like to thank the personnel and the administration of the University of Ioannina, Computer Science Dept. and the EC-JRC ISIS for their support during this research.

The views expressed by this work are purely those of the writer and may not in any circumstances be regarded as stating an official position or the views of the European Commission and the Joint Research Center.

References

A Study of the Behavior of Equivalent Algorithms in Fault Injection Experiments


15. C. Moler, C. Van Loan “Nineteen dubious ways to compute the exponential of a matrix” SIAM Review Vol. 20 No. 4, October 1978 pg. 801-836


The Effectiveness of Statistical Testing when Applied to Logic Systems

Silke Kuball\textsuperscript{1}, Gordon Hughes\textsuperscript{1}, John H.R. May\textsuperscript{1}, Julio Gallardo\textsuperscript{1}, Andrew D. John\textsuperscript{2}, and Roy B. Carter\textsuperscript{2}

\textsuperscript{1} Safety Systems Research Centre, Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol BS8 1UB, UK, Silke.Kuball@bristol.ac.uk
\textsuperscript{2} NNC Ltd., Booths Hall, Chelford Road, Knutsford, Cheshire WA16 8QZ, UK, Andrew.John@nnc.co.uk

Abstract. In this paper we demonstrate the effectiveness of statistical testing for error detection on the example of a Programmable Logic System (PLS). The introduction of statistical testing arose from the wish to quantify the PLS’s reliability. An appropriate statistical testing algorithm was devised and implemented, which is described in detail in this paper. We compare the results of statistical testing with those of a variety of other testing methods employed on the PLS. In terms of differences detected per number of tests, statistical testing showed an outstanding effectiveness. Furthermore, it detected a problem, which was missed by all other testing techniques. This together with its potential for reliability quantification illustrates its importance for system validation as part of a risk–based safety–case.

1 Introduction

The replacement of existing and obsolescent safety or safety related systems has become an important issue, for example in the nuclear industry. It potentially involves significant economic risks for plant/system operators. This is why recent research within the Nuclear Safety Research Programme, part of the UK Health and Safety Executive Research Programme has had among others the objective to independently test a provided replacement system in order to confirm that the targets are achieved. The work described in this paper is based on a hypothetical replacement for the discrete (electronic hardware based) safety interlock system on an Advanced Gas Cooled Reactor (AGR) charge machine. This paper reports in detail on the statistical testing approach developed to estimate the reliability of the replacement system. Besides providing the basis for reliability quantification - something other testing techniques do not provide - this approach convinces through its effectiveness in terms of fault finding when compared to the other testing techniques employed. We believe that the general features of the introduced statistical testing approach can be reused on other sequential control systems in a variety of industrial sectors. We start by introducing the replacement system and the test-equipment setup in section 2. Different testing
strategies applied during system assessment are then described. In section 3, the statistical testing technique is described in detail. Section 4 compares the different testing techniques with regard to their effectiveness in terms of fault detection.

2 Replacement Legacy System Experiment

The GEC Elliot Logicon 2 logic system employed on the Hunterston B Charge Machine has been taken as an example of a typical legacy system, which may require replacement at some point. A sub-set of the logic associated with the turret rotate function, containing turret rotate interlocks, turret rotate control and indicators was selected for the experiment. The interlocking logic is straightforward, being “permissive energise to go”. The indication logic is also straightforward; the indications reflect the plant state derived from the logic and plant state inputs. The control logic contains latch circuits as part of the turret drive control, cross interlocks to prevent simultaneous anti clockwise and clockwise drive demands, and speed control logic to switch between fast, slow speeds and stop as a position is approached. The logic sub-set uses 50 binary inputs (6 classified as being from safety devices) and 14 outputs.

One full and failure-free cycle of system operation is referred to as the normal operating sequence. It represents what one would typically observe under normal plant operation. In this study we considered the example of one full cycle of fuel exchange, which was identified to be a sequence of 158 50–bit input strings. Other operational sequences are possible and can be easily accommodated in the studies described here.

A typical supplier of high-integrity safety equipment was selected to design and produce a replacement system using their DSP Programmable Logic System (PLS). The produced PLS system was subsequently modified in the light of factory tests and user comments to produce the version, which forms the subject of the experimental evaluation reported below.

In an independent exercise NNC Ltd. designed a test “oracle” for the logic sub-set. This was done from the plant drawings using the logicon users guide to produce a functional representation of the logic i) in “AND/OR” logic form and ii) in programmed plus logic functions using $C^{++}$ code. The functional logic was implemented on a commercial Programmable Logical Controller (PLC) to produce a physical test oracle for experimental purposes. The $C^{++}$ code was only used for off-line checks. Test schemes were produced and generated on a test control PC. The PC sends input state test sets to both the PLC and PLS. It reads back the output states generated by each of the logic implementations, compares them with the expected state and checks for consistency and valid outputs. The interconnection between PC, PLC and PLS can be seen in Fig.1. A range of testing techniques have been implemented using this arrangement including:
1) **Single Random Input Tests.** Starting with an initial 50-bit input string, a single bit is changed at random in each step. Each step results in a new test-case.

2) **Total Random Tests.** 50 bit input strings are generated completely at random with each bit taking on the values 0 or 1 with probability 0.5 respectively. Each generated input-string forms one test-case.

3) **Plant Simulation Tests.** One full fuel-exchange cycle consisting of 159 input strings (1 to take the system back to initial state) is run several times with fixed and varying time delays between inputs.

4) **Interlock Total Random Negative Tests.** These are similar to the total random tests with one bit (an interlock) fixed to either 0 or 1 throughout the tests.

5) **Targeted Sequence Random Tests.** A test is made up of a series of inputs that follows the identified normal operating cycle for a number of steps and then allows a limited number of input bits to be randomly changed.

6) **Statistically Valid Tests.** Statistically independent tests are generated that represent simulations of the actual operational environment.

The algorithm devised for statistically valid testing and the results achieved with it form the focus of this paper.

### 3 Statistically Valid Testing

#### 3.1 Background

Any logic or program can contain systematic faults, which can remain undetected for most inputs and only induce system failure in very rare cases. These “exotic cases” can however be linked to input scenarios, on which failure would have
hazardous consequences. Since not all potentially possible inputs can be tested, one has to find another way of establishing confidence in the system. Statistically valid testing or simply statistical testing exposes the system to a simulation of its operating environment, thereby simulating normal input-scenarios as well as error-scenarios. Statistical test-cases have to be independent samples from a probability distribution, the operational profile. The operational profile describes the likelihood of encountering certain scenarios in actual operation at any given point in mission time. From the results of statistical testing one can quantify system reliability by estimating the probability of the system to fail when demanded to act at any point in mission time, the pfd. In a first general advice document on the underlying study, it was established that 46,050 statistical tests run on the PLS without revealing failure would indicate a pfd of less than $10^{-4}$ to a 99% confidence limit. Remark: This number holds for statistical testing only. From other forms of tests, no such conclusion can be derived. Some mathematical background on statistical testing can be found in [1], [2], [3], [4].

In the next two sections, we describe the statistical testing algorithm devised for the PLS under study. This algorithm was implemented in Visual C++ to form a tool (StatTCG) for automatic statistical test-case generation. Sets of test cases were produced as Excel–spreadsheets to be compatible with NNC’s test-equipment.

3.2 Statistical Test Inputs for the PLS

Input to the PLS consists of 50 binary values reflecting plant–state or the state of push–buttons. Thus the total set of inputs, the input space consists of $2^{50}$ input strings. A 50–bit input string that is part of the normal operating sequence is called normal input string. Based on discussions with the provider and our collaborators, we formulated the assumption that each execution of the PLS is an independent calculation of the output states. External registers are present that allow information to be passed between execution cycles, but they form part of the input string and can thus be accessed and explicitly modelled through the input generation mechanism. Therefore they do not constitute any hidden effects that might affect independence. As a result, a statistical test–case is a single 50–bit input string.

Statistical testing requires the definition of an operational profile, see also [5], [6]. In the following we establish an operational profile by using a physically meaningful partition of the system input-space into a set of bins. We produce statistical test cases as deviations from the normal operating sequence. These are the result of errors occurring in the plant or its environment, and they take on the physical form of bits in an input string being in the “wrong” state, i.e. being switched off when they should be “on” and vice versa. We assume that more than one deviation can occur at the same time. The operational profile is modelled in three layers. The first layer is a probability distribution describing the number of deviations occurring at the same time within a normal input

---

1 One full and failure-free cycle of system operation.
string. We define a set of input-space bins as Bin \( k := \) “Occurrence of exactly \( k \) deviations simultaneously within a normal input string”, \( k \geq 0 \). An element from Bin \( k \) is called a \( k \)-order deviation from normal. This can be understood as “\( k \) things have gone wrong” at the same time at some point in the operational cycle. Obviously a 0-order deviation is a normal input string. The following distribution over the set of bins was chosen. Let \( p \in [0, 1] \) be the probability of a deviation to occur on any given input string.

\[
Pr(\text{Bin } k) = p^k \cdot (1 - p), p < 0.5.
\]  

Fig. 2 shows a plot of the distribution in (1) for the example \( p = 0.05 \). \( p \) can be set on the StatTCG interface. Eq. (1) is based on the assumption that on an input

\[
\begin{array}{c|c|c|c|c|c|c}
\text{Bin number} & \text{Number of deviations} & \text{Bin probability} \\
\hline
0.05 & 1 & 0.95 \\
0.2 & 2 & 0.8 \\
0.4 & 3 & 0.6 \\
0.6 & & 0.4 \\
0.8 & & 0.2 \\
0.95 & & 0.05 \\
\end{array}
\]

Fig. 2. Probability distribution over input–space bins.

...
For deviations of type 1. and 2. above, we assume the probability 0.45 respectively. For deviation of type 3., the assumed probability is 0.1. These are constructed probabilities chosen in this model in the absence of more specific information. These probabilities can be changed in StatTCG as soon as new data on them become available. Given that a deviation occurs and given that it is of type x, x=1,2,3, then in the third layer of our operational profile we randomly pick from the set of all possibilities those single bits or multiple-bit switches that are to be inverted.

### 3.3 Algorithm for Statistical Test Case Generation

The set of 0-order deviations (the normal operating sequence itself) was tested before statistical testing started. The set of 1-order deviations is small, it contains 9480 elements. Thus the system can be tested exhaustively on this set. StatTCG contains a first part producing the full set of first-order deviations. The PLS is tested once on this set. The actual statistical testing part of StatTCG focusses on 2nd and higher order deviations. The fact that we exhaustively test Bin 1 and only consider Bin 2 or higher for statistical testing contributes to a high effectiveness of our statistical testing technique. The algorithm implemented in StatTCG can be summarized as follows.

1. Randomly pick one input string from the set of 158 normal 50-bit input strings. Each single input string is picked with the same probability $\frac{1}{158}$.
2. Pick the deviation type (1,2 or 3) according to the probabilities specified above.
3. If deviation type 1 is chosen, one of a list of identified “single-bit switches” is randomly picked and its state reversed. If deviation type 2 is chosen, a “multiple-input switch” is randomly picked from the set of all identified “multiple-input switches” and the state of all group members is reversed. Analogously one proceeds in the case of deviation type 3. Thus a new 50-bit string is created.
4. Go back to 2 and perform another deviation on the input string created in 2 and 3.
5. Generate a uniform random variable $X$. If $X < p$, insert another deviation as in steps 2 and 3. $p$ is the probability from Eq. (1). Repeat step 5. If $X \geq p$, write the generated input string into the test case file. Go back to 1. Repeat until a set of $N$ test cases has been generated. $N$ is specified by the tester on the StatTCG interface.

The bulk ($\sim 95\%$) of all statistical test cases generated with StatTCG, using $p = 0.05$, lies in a close environment of the normal operating sequence. There is the possibility to encounter more exotic cases representing the occurrence of more than two problems occurring at the same time, but this will be a small percentage ($\sim 5\%$). This is in agreement with the assumption that such exotic input situations occur most infrequently in actual plant-life. The most frequently encountered situations are either one particular issue occurring or two things
going wrong at the same time or shortly after each other. The three major features of the statistical testing algorithm implemented in StatTCG are:

1. It is built around an identified normal operating sequence.
2. It focusses on physically meaningful input strings. These are input strings that represent physical conditions in the plant or its environment, which result in the setting or unsetting of switches.
3. Other, more exotic cases of input are not excluded, but occur with very low frequency.

4 Results from Testing

In this section, we compare the results obtained when testing the final version of the PLS code, Version 4.0, with the testing techniques described in sections 2 and 3. The initial factory tests and review processes had discovered a number of errors in Version 3.0, which were eliminated in Version 4.0 and will not be discussed. During testing of the final version, Version 4.0, differences in the PLS and PLC output occurred that were traced back to three errors:

**Error 1:** PLS “no drive” but PLC oracle “drive”. This is due to a known difference between the two implementations.

**Error 2:** PLS “drive” but PLC oracle “no drive”. This is due to timing issues (high speed (1msec) cycle time of the PLS).

**Error 3:** This is an oscillating effect that keeps both clockwise and anticlockwise rotation energised. It appears to involve an interaction between the PLS diagnostic output shutdown logic and the application logic clockwise/anticlockwise interlocking. It is currently being investigated by the supplier.

We start by describing the results of the non-statistical tests, 1)-5) and then describe the results of statistical testing. Some of the non-statistical tests contain a random element, however this element is not based on any model of actual operational use. As can be seen in the results this induces a low efficiency in error detection when compared to statistical testing.

**Results with non-statistical testing:**

1) **Single Random Input Tests. Result:** A total of 520,000 single random input tests were produced. 57 differences occurred, which were traced back to Error 1. **Remark:** The single random input tests lead gradually further away from the initial input string and lead us into an area of the input space where inputs lie that are physically meaningless. They are unlikely to represent conditions, on which the PLS will actually be challenged because they are too obviously distorted.

2) **Total Random Tests. Result:** A total of 990,000 tests were run producing 157 differences, of which 127 were traced back to Error 1 and 30 to Error 2.
Remark: Again, these tests do not take into account what the actual operating sequence is and how “far away” from the normal operating sequence the generated string is. They seem to be too unrealistic to be effective.

3) **Plant Simulation Tests. Result:** These tests were run 378 times with fixed time delays between inputs and twice with varying time delays. No differences occurred.

4) **Interlock total random negative tests. Result:** A set of 520,000 tests was run. 31 differences occurred, of which 29 were traced back to Error 1 and 2 were traced back to Error 2. **Remark:** Same as for total random tests.

5) **Targeted Sequence Random Tests. Result:** A set of 30,000 tests produced 57 differences, of which 12 were due to Error 1 and 45 were due to Error 2. **Remark:** These tests are taking into account the normal operating sequence, they already seem to be more efficient. However they do not focus on first- or second-order deviations and do not they take into account what physically meaningful deviations are.

**Results from statistical testing with StatTCG:**

6) a. **First order deviations.** The full set of 9480 1–order deviations was applied on the PLS code. 432 differences were observed. These were traced back to Error 1.

6) b. **Higher–order deviations.** These are the actual statistically produced tests representing second- or higher–order deviations from the normal operating sequence. 57,500 tests were generated and applied to the PLS code. 428 differences and 5 invalid outputs were observed. These were traced back to Error 1 (155 times), Error 2 (273 times) and **Error 3** (5 invalid outputs), which had been so far undetected.

**Remark:** It should be noted that none of the errors identified provided evidence that the logic did not implement the required safety functions correctly.

4.1 Comparing Effectiveness

To visually compare the effectiveness of the different testing methods, we define the “Test effectiveness of test method $M$ for error $E$”:

$$T\text{-eff} (M, E) := \frac{\text{Number of detected differences with } M \text{ due to } E}{\text{Number of tests performed with } M}.$$  

Fig. 3 contains a plot of $T\text{-eff}(M, E) \times 10^5$ for test methods 1)-5) and 6) b. above with respect to Errors 1, 2 and 3. Fig. 3 is plotted on a logarithmic scale. Alongside we have plotted the total effectiveness of each testing method as the total number of differences detected divided by the total number of tests performed with that method, again multiplied by $10^5$.

**Result:** Statistical testing stands out in performance with regard to error detection effectiveness. Not only did it detect more differences in a smaller test set, it also detected an effect (Error 3), which none of the other performed testing methods detected.
Remark regarding the aim of reliability estimation: The initial aim of introducing statistical testing was the quantification of PLS reliability. Due to the high number of inconsistencies found and the detection of an unwanted oscillation effect, this would currently be inappropriate. Formulae to estimate the probability of failure on demand on the software under test based on the results of statistical testing can be found in for example [1], [2], [3]. However, only after the occurred differences have either been removed or classified as tolerable should reliability quantification be considered.

5 Conclusion

The high effectiveness of a statistical testing technique compared with other testing methods has been demonstrated for the example of a programmable logic system.

It can be argued that the high effectiveness is achieved because the statistical tests concentrate on those parts of the input space that are physically meaningful in terms of plant failure conditions. The introduced algorithm produces effects simulating the occurrence of several problems occurring at the same time under various circumstances, but always closely associated with the normal operating sequence. This makes it very relevant for the detection of unwanted properties or even faults in a given implementation. Thus, statistical testing - if properly designed - constitutes a rich testing environment that has the chance to actually trigger the kind of problems a real-world system can be exposed to. In this case,
reliability estimation from statistical testing is based on a “realistic” observation of system performance in actual operation. It appears that the occasionally heard concern that statistical testing “does not find faults”, should be reconsidered. If properly designed, statistical testing can even find problems that other testing strategies miss. This together with its potential for reliability quantification makes it a very important testing method that should be considered as a worthwhile element of a safety–case.

Acknowledgement: This work was funded under the HSE Generic Nuclear Safety Research Programme and is published with the permission of the Industry Management Committee (IMC). The authors would like to thank the staff of British Energy and the Nuclear Installations Inspectorate involved in the New Development of Dynamic Testing (NewDDT) and SIARLS projects.

IPR statement: The information contained in this report has been produced for BEG/BEG(UK) Ltd./BNFL/Magnox Electric plc. on behalf of the Industry Management Committee (IMC) thus is the joint property of British Energy Generation Ltd., British Energy Generation (UK) Ltd, British Nuclear Fuels Ltd., and their successor companies. This information is to be held strictly in confidence by the originators and recipients, unless the contract specifies otherwise. No disclosure is to be made to a third party without the written agreement from the nominated officer from the company that placed the contract or their nominated deputy. Further any Intellectual Property Rights arising from or contained in the report are joint property of British Energy Generation Ltd., British Energy Generation (UK) Ltd., British Nuclear Fuels Limited and their successor companies, as such are subject to the same obligation of confidence on the originator and recipients as set out above and may only be used in accordance with the contract.

References

A Classification Scheme for Software Verification Tools with Regard to RTCA/DO-178B

Günther Zoffmann, Manfred Gingerl, Christian Reumann, and Gerald Sonneck
ARCS, A-2444 Seibersdorf, Austria
{manfred.gingerl, christian.reumann, gerald.sonneck, guenther.zoffmann}@arcs.ac.at

Abstract. A current goal of ARCS is to set up a software lab dedicated to software verification in accordance with international recognized guidelines and standards, e.g. RTCA/DO-178B. Investigations resulted in a large number and variety of commercially available software tools. A practicable classification scheme for software verification tools was needed. This paper provides a tool classification scheme that is based on four basic tool classification categories: “objectives”, “methods”, “metrics”, and “attributes”.

1 Introduction

The need for a classification scheme for software verification tools arose during the planning process for a software lab dedicated to verification activities. The intention of this project is to utilize the software verification lab finally for activities within the framework of software certification for airborne systems according to RTCA/DO-178B [1] which was used as basis for this work. An objective in the first phase of this project was to evaluate commercial test and analysis tools, i.e. to assess usability and overall quality of specific products and to figure out possibilities and limits of commercially available techniques and methods.

The investigation and comparison of a number of currently offered software tools resulted in a large variety of products with an even larger number of techniques, methods, tool attributes and properties. Soon it turned out that many of the software tool producers and distributors stated their own special tool categories – most likely for marketing reasons.

The problem with such a diversity of tool categories is that these are often incomparable so that it is nearly impossible to classify the tools, e.g. for structured storage in a database for later evaluation. This is the reason why a practicable classification scheme for software verification tools was developed.

The first question to answer was: How can software verification tools be defined in contrast to other software tools? In general a software tool (for use within a software life cycle) is “a computer program used to help develop, test, analyze, produce or modify another program or its documentation” [1]. Regarding this basic definition software verification tools can be defined as software tools used to automate, reduce or even eliminate well-defined software verification activities, which are typically combinations of analyses, tests and reviews (including formal inspections [2]).
To point out the difference between software verification tools and software development tools it is helpful to refer to goals of the related life cycle processes.

As illustrated in Fig. 1 the software verification process is just one of four software integral processes stated by RTCA/DO-178B to “ensure the correctness, control, and confidence of the software life cycle processes and their outputs” [1].

While the software development processes produce the software product, the purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes. Unlike software development tools, software verification tools cannot introduce errors but may fail to detect them.

Fig. 1. Software life cycle processes in accordance with RTCA/DO-178B [1]

This paper is organized in 7 sections. Basic categories for a tool classification scheme are introduced in section 2. Subsequent sections 3 to 6 should be understood as a classification approach for each tool category, i.e. an overview of most common techniques, methods, metrics and properties or attributes of software verification tools. Section 7 contains a short summary.

2 Basic Categories for a Tool Classification Scheme

A classification scheme for software verification tools should take into account the full variety of functional, methodical, technical and also practical aspects. On the other hand, if the classification complexity exceeds a certain degree, the classification scheme will most likely miss its goal.

The best and easiest way out of this dilemma is by considering that only certain combinations of tool classes exist (e.g. objectives and methods or methods and metrics), and that not all tool classes that can be stated on demand, represent commercial products that are actually available. For example, some kind of
maintainability metrics would sometimes be useful, but it does not make sense to state this class, if such tools are not available.

The subsequent listing of crucial questions related to software verification tools can be utilized to define four basic tool categories (TC) as a starting point for a practicable tool classification scheme as illustrated in Fig. 2:

**TC1: “OBJECTIVES”**
What is the declared verification objective or the intended purpose of a specific tool?
It is obvious that verification objectives of software tools in general do not tell much about provided methods and techniques, implemented metrics or specific tool properties. Nevertheless it is essential to know the intention behind a software tool and its dedicated purpose of usage within the software verification process.

In any case the verification objective of a specific software verification tool (declared and provided by the producer or distributor of that product) must comply with requirements of the particular software life cycle phase.

A common example for a verification objective is regression testing. The purpose of regression test tools is to demonstrate that program extensions or source code modifications do not influence existing functionality or performance of the software product in concern. It is evident that such a task is not bound to a specific testing technique or method.

Other examples are: reliability assessment, test coverage analysis.

**TC2: “METHODS”**
Which verification techniques and methods are provided to achieve those objectives?
Thorough specification of provided techniques and methods of a software verification tool is an essential classification goal. It tells the user how the task to achieve an intended verification objective is performed by that specific product. Of course this information does not tell much about verification objectives and therefore techniques and methods have to be assessed and classified by TC1 (and others) as well.

Fault injection is an example for a common software testing technique. It works by causing faults, forcing exceptions, and stressing an application to test overload conditions. This testing technique can be utilized for verification of software with different intentions: software reliability assessment or test coverage improvement.
TC3: “METRICS”
Which software metrics are implemented?
Software metrics are well-defined quantitative measures of extent or degree of a certain software characteristic, quality, property or attribute. Within the software verification process software metrics can be utilized for different purposes, e.g. for quantitative assessments of verification results (e.g. for coverage analysis) or as a basis for other software metrics.

Common examples are: lines-of-code metrics, complexity metrics.

TC4: “ATTRIBUTES”
By which attributes, properties, quality aspects and features is a tool characterized?
This category includes usability aspects as well as technical attributes of software verification tools. The latter are relevant in particular when purchasing a commercial product for certain intended verification tasks within special hardware and software environments. This category is discussed in more detail in section 6.

Common examples are: tool qualification (for certification), compatibility issues (with operating systems), automation features.

3 TC1 Classification (OBJECTIVES)

TC1 classification provides classes of software verification objectives (as stated in the previous section). As illustrated in Fig. 3 TC1 classes are related to high-level objectives, e.g. portability, as well as to low-level verification objectives that are derived from those high-level goals, e.g. source code compliance.

The so-called “dependability-explicit model” [4], a promising new development model that generalizes the corresponding RTCA/DO-178B model, provides a useful set of verification objectives according to four stated dependability processes.

In general high-level verification objectives are characterized by abstract goals like overall quality aspects of software while low-level objectives are more specific. In any case, verification objectives must be related to software requirements or applicable software quality guidelines and standards.

![Fig. 3. TC1 high-level and low-level classification (objectives)](image-url)
3.1 High-Level TC1 Classification

Software quality characteristics like those defined by ISO/IEC 9126 [3] can suitably be utilized as a basis for a high-level TC1 classification: functionality, reliability, usability, efficiency, maintainability, portability. Although this basis may not be complete it is a good starting point for further extensions and refinements.

3.2 Low-Level TC1 Classification

The following sub-sections describe common verification objectives that are derived from software quality objectives.

3.2.1 Code Compliance

Compliance of source code with coding standards can be related to product specific coding requirements or to applicable software guidelines and standards (e.g. ANSI, IEEE, IEC). Code compliance meets three important verification objectives:

- to improve reliability (reduce the probability of programming errors)
- to improve maintainability
- to improve portability (reduce compatibility issues and improve reusability)

3.2.2 Software Performance

Software requirements typically include verification objectives related to software performance. Such objectives (e.g. certain response times) are very important for time-critical applications, in particular when processing huge data files.

3.2.3 Test Coverage

Coverage analysis related to software is the “process of determining the degree to which a proposed software verification process activity satisfies its objectives” [1]. Coverage analysis related to testing activities is known as test coverage analysis. The primary objective of test coverage analysis is to quantify the extent of coverage. The process of test coverage analysis comprises functional and structural test coverage:

**Functional Test Coverage**

Functional coverage analysis is also known as requirements-based coverage analysis because test cases or testing procedures are analyzed in relation to the specified software requirements. The objective of functional coverage analysis is to determine to which degree the set of requirements-based test cases verified the implementation of the software requirements [1].

**Structural Test Coverage**

Structural coverage analysis is a common software verification technique to analyze the degree to which a code structure has been exercised by given test cases or test procedures. Another usual term for structural coverage analysis is code coverage analysis. Common measures are listed in section 5.
4 TC2 Classification (METHODS)

There is a large variety of special techniques and methods for all kinds of software verification objectives, which cannot be presented within the limits of this paper. The most common techniques and some well-established methods are stated below as typical examples:

*Code Compliance Analysis*
Code compliance analysis is a static software verification technique of checking source code for compliance with applicable coding standards (e.g. ANSI) and specific coding requirements involving syntactical and structural programming rules.

*Unit Testing*
The technique of testing the individual subprograms, subroutines or procedures in a program is called unit testing.

*Integration Testing*
Integration testing is the process of progressively aggregating individual system components to demonstrate proper interaction. Typical problems identified are improper call or return sequences or inconsistent handling of data objects.

*Fault Injection*
Fault injection techniques work by causing faults, e.g. by providing input data out of specification or by forcing exceptions.

*Stress Testing*
Stress testing techniques stress applications to test overload conditions.

*Robustness Testing*
RTCA/DO-178B [1] suggests black-box test cases to demonstrate the robustness of software, i.e. the ability of the software to respond to abnormal input and conditions.

5 TC3 Classification (METRICS)

Many different software metrics are implemented in software verification tools for support of the software verification process and thus can be classified. A few common examples are discussed here.

5.1 Code Coverage Measures

A large variety of code coverage measures exists. The basic and most common measures are listed below.
Function Coverage
Function coverage is a simple measure to get coverage information about functions or procedures invoked by a test. It is useful during preliminary testing to assure some coverage in all areas of software.

Call Coverage
Call coverage relies on the hypothesis that many faults are related to interfaces between software units (functions, procedures, modules, …). The call coverage measure reports whether each call of a software unit was executed.

Statement Coverage
The function of statement coverage is to report the portion of executable statements of source code or object code encountered by applying a certain test case or test procedure. Statement Coverage is also known as line coverage.

Basic Block Coverage
Basic block coverage is essentially the same as statement coverage except that the unit of measurement is a sequence of non-branching statements.

Decision Coverage
Decision Coverage reports whether entire (logical) expressions in control structures (such as if-statements and while-statements) have been tested with all possible results. This measure includes coverage of switch-statement cases, exception handlers, and interrupt handlers.

Condition Coverage
This measure is very similar to decision coverage. The difference is that condition coverage takes the true or false outcome of each boolean sub-expression (separated by logical operators) into account.

Path Coverage
The path coverage measure reports whether each of the possible paths in each function has been followed. A path is a unique sequence of branches from the function entry to the exit.

5.2 Code Metrics

Common source code metrics are [2]:
- Lines of Code (LOC): a number, generated by counting the lines of source-code (without counting comments)
- McCabe Cyclomatic Complexity Metric: uses flow structure of a program as relative measure of its complexity
- Halstead’s Software Science Complexity Metric: measures complexity based on the program’s size in terms of operators and operands.
6 TC4 Classification (ATTRIBUTES)

6.1 Miscellaneous Attributes

Static / Dynamic Software Verification (ST / DY)
A meaningful classification of software verification tools is given by distinction between static and dynamic software verification techniques.

Static software verification denotes technical assessments as well as measuring and evaluation activities to verify software without running its executable code. Static software verification tools are also identified as static analysers. Common examples are code compliance analysis tools and structural coverage analysis tools.

Dynamic software verification techniques and methods are characterized by the fact that the target of evaluation has to be executed to perform the intended verification activity. Dynamic software verification tools are also called test tools. Examples are performance test tools, integration test tools and regression test tools.

Deterministic Software Verification
Results of deterministic software verification tools are reproducible, i.e., deterministic tools produce the same output for the same input data when operating in the same environment. This is a necessary condition for qualification of a tool.

Qualified Software Verification
For compliance with certification regulations or safety-related software standards and guidelines like, e.g., RTCA/DO-178B [1] tools must be qualified for use within the software verification process. Any software tool used for automation, reduction or elimination of software verification process activities has to be qualified.

Automated Software Verification
The use of software tools to automate verification activities within the software life cycle processes can help satisfying dependability objectives insofar as they can enforce conformance with software development standards.

6.2 Compatibility Classes (COMP)

In general a classification for software and hardware compatibility attributes has to be specified for the software verification tool itself and for the target of evaluation, i.e. the software object under investigation. This depends on the specific verification activity of that tool.

In case of static software verification activities the target of evaluation (e.g. the source code) is independent of the verification environment (hardware and software). This is due to the fact that its software code does not have to be executed for static verification activities, e.g. source code analysis.

A more complex situation is given in case of dynamic software verification tools, where code of the target of evaluation has to be executed by definition.
Software Compatibility Classes (SW-COMP)
Software compatibility classes provide compatibility information related to three
types of environments:
- software development environment
- software test environment
- software runtime environment

It is practicable to define software runtime environment compatibility classes like
VMS, Windows/32, Unix, etc. and sub-classes like Windows NT, Windows 2000,
HP-UX, Solaris, etc. Sub-sub-classes specify version information related to the
software environment (e.g. Windows NT 4.0 Service Pack 3)

Hardware Compatibility Classes (HW-COMP)
Hardware compatibility classes identify tool-compatible hardware environments, i.e.
microprocessors, storage devices, network connections, input and output devices,
customized hardware, etc.

6.3 Software Coding Classes (CODE)
Software coding classes (related to targets of evaluation) represent all kinds of
software code, i.e. executable object code, interpreter code and source code, i.e.
programming languages, e.g. ADA, C, C++, Fortran77, …).

7 Further Work
This paper presents a kind of snapshot in ongoing research in the field of software
tool classification currently performed at the Austrian Research Centers Seibersdorf.

The classification of tools related to verification and validation was identified as
crucial step for the proper handling of both customer request and purchasing
decisions.

The next steps are to practically apply the classification scheme to real product
descriptions on the one hand and, on the other hand, to develop a “decision tree” that
allows an interactive tool selection based on the customers needs.

Also the requirements related to other standards than RTCA/DO-178B (as there
may be IEC 61508, ISO 15408, etc.) shall be taken into account.

8 Summary
This paper suggests a classification scheme for software verification tools that is built
on four basic categories: “objectives”, “methods”, “metrics”, and “attributes”. The
intention was to provide a generic basis for a meaningful classification of software
verification tools that is also practicable. Following this classification approach
ensures that verification objectives of a classified software tool are well-defined and
cannot be confused with verification techniques or methods. Such a classification approach is compliant with guidance and terminology of RTCA/DO-178B [1].

Regarding the diversity of software verification activities and environments it would be impossible to present a complete overview of software verification techniques and methods or to give a detailed list of all known software metrics and tool attributes within the framework of this paper. Therefore the set of included examples (sections 3 to 6) should be understood as a baseline for further refinement.

Typical examples for a tool classification are presented in the following tables (both representing existing commercial products).

**Table 1. Software verification tool 1**

<table>
<thead>
<tr>
<th>TC1-H</th>
<th>portability, maintainability, reusability</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1-L</td>
<td>code compliance</td>
</tr>
<tr>
<td>TC2</td>
<td>(requirements-based) code review, coding style analysis</td>
</tr>
<tr>
<td>TC3</td>
<td>cyclomatic complexity, Halstead metrics</td>
</tr>
<tr>
<td>TC4</td>
<td>automated static software verification; deterministic, unqualified</td>
</tr>
<tr>
<td>TC4-COMP</td>
<td>Unix (AIX, HPUX, IRIX, Solaris), VMS, Windows (9x, NT/2000)</td>
</tr>
<tr>
<td>TC4-CODE</td>
<td>C, C++</td>
</tr>
</tbody>
</table>

**Table 2. Software verification tool 2**

<table>
<thead>
<tr>
<th>TC1-H</th>
<th>reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1-L</td>
<td>stress testing</td>
</tr>
<tr>
<td>TC2</td>
<td>source code analyses; test execution with random and incremental test patterns; unit testing</td>
</tr>
<tr>
<td>TC3</td>
<td>-</td>
</tr>
<tr>
<td>TC4</td>
<td>automated, static and dynamic, software verification; deterministic, unqualified</td>
</tr>
<tr>
<td>TC4-COMP</td>
<td>MS-DOS, Windows (3.x, 9x, NT/2000)</td>
</tr>
<tr>
<td>TC4-CODE</td>
<td>Ada, (C)</td>
</tr>
</tbody>
</table>

**References**

Safety Patterns – The Key to Formal Specification of Safety Requirements

Friedemann Bitsch

Institute of Industrial Automation and Software Engineering, University of Stuttgart
Pfaffenwaldring 47, 70550 Stuttgart, Germany
Phone: +49 711 685 7292, Fax: +49 711 685 7302
mailto:bitsch@ias.uni-stuttgart.de

Abstract. The use of formal methods increases the trust in the safe operation of software in industrial automation systems. But the use of formal methods in practical software development is rare. One of the reasons lies in the difficulties arising from formal specification of safety requirements by common software engineers who are not experts in logic. In this paper an approach is presented, in which the difficulties are overcome by the use of formal specification patterns. The main advantage in comparison to other approaches is that the specification patterns transfer expert knowledge. Therefore this approach not only helps in using formal methods, it also supports in learning the practical application of formal specification languages for safety requirements specification. The patterns are called "safety patterns" because they are developed for the formal specification of requirements special in context of safety.

1. Introduction

Business competition and new developments in technologies have led to an improvement not only in the capability characteristics of industrial automation systems but also to a higher complexity of the systems. This implicates higher fault vulnerability in the development of systems. Especially in systems with safety responsibility, faults must not occur, because faults could not only lead to high costs but could also cause high material damages and personal injuries. Nowadays software control in industrial automation systems is common. But especially in software development for complex systems faults in development often occur.

By the use of formal verification, trust in the safe operation of software can be increased, compare [13]. The goal is the formal proof that an operational software model fulfils safety requirements. For this the specifications must be in a formal notation. This means that the notation has a well-defined and precise syntax and semantic. [15]: "Formal specification of the required system behaviour has many benefits. For example, a formal specification is unambiguous and precise, [...] and can help ensure that a specification is well-formed." It would be obvious to use formal

1 This work was sponsored by the German Research Council (DFG) within the scope of the focus area program (1064) on the "Integration of Specification Techniques with Applications in Engineering"
verification to check a model against an entire requirements specification. But from an economical view this would be too expensive, because formal verification still requires too much man power. Therefore the use of formal verification is only reasonable for safety critical parts of industrial automation systems.

"But the use of formal methods in practical software development is rare. A significant barrier is the widespread perception among software developers that formal notations and formal analysis techniques are difficult to understand and apply", see [16].

We believe that the formal verification method of model checking contributes to overcome some of these problems. It is a finite-state verification method in which the formal check of the operational model is performed automatically in contrast to theorem proving, which requires guidance by an expert in formal methods. Using model checking, algorithms pass through the complete state space of the operational model of the software and simultaneously the compliance of the requirements specification is checked. For that purpose the operational model must be specified as a finite-state transition system, while the requirements are typically specified with Temporal Logic, see [10]. Model checking is an alternative to theorem proving for those cases in which the state space does not "explode" at the calculation of all combinations of consecutive states. This depends on the structure of the model and the number and the value area of the variables. There exists several techniques to avoid state space explosion, compare [5]. Nowadays there are very efficient model checkers like the freely available SMV or VIS, see [27] and [32]. These model checkers use an implicit "symbolic" representation of state transitions and state labelings. This enables the verification even of complex operational models.

Despite the automation of model checking, the user still must be able to specify the safety requirements with a formal specification language, which is realised in any kind of Temporal Logic. In [17], the resulting difficulty is described: "Expressing certain properties in Temporal Logic is complex and error-prone, not only for practitioners but for formal methods experts as well.” A reason is stated in [18]: "[...]

The model checkers VIS and SMV need requirement specifications to be written in the Temporal Logic CTL (Computation Tree Logic). This logic has proven to be extremely fruitful in verifying hardware and communication protocols; and software developers are beginning to apply it in the verification of software, see [19]. In specifications in CTL, the temporal order defines a tree, which branches towards the future.

CTL formulas consist of particular propositions. Every proposition corresponds to variables in conditions, events and actions of the operational model. The propositions are related by standard connectives of Propositional Logic and CTL temporal connectives. Connectives of Propositional Logic are AND, OR, XOR, NOT, and ≠. Every CTL temporal connective is a pair of symbols. The first symbol is a path quantifier. In calculating the state space there are many execution paths starting at the current state. The symbol is one of A and E. A means "along All paths" and E

2 "p ⊨ q" stands for implication between p and q, suggesting that q is a logical consequence of p.
means "along at least (there Exists) one path". The second pair is one of temporal modalities, which describe the ordering of propositions in time along an execution path. These are X, F, G or U, meaning "next state", "some Future states", "all future states (Globally)" and "Until", see [19] and [32]. An example of a CTL formula is (1). Such kind of formal formula is difficult to read, to understand and to write correctly for an engineer, who is not an expert in formal methods. It easily happens, that a formula is specified, which state something different to which should be expressed.

\[
AG (a \rightarrow EX ((EF b) \land EF (c \rightarrow AG \neg b)))) \lor AG (\neg b)
\]

(1) means that the proposition b may be valid only after a has occurred and only before the condition c is not valid.

By specifying safety requirements in a formal notation there are the following difficulties for software engineers:

1. The expression of the requirements in the context of the operational software model.
2. The use of formal languages itself.

The first difficulty is discussed in [28], [2] and [4]. Our approach is, that by using techniques of safety analysis like Fault Tree Analysis (FTA) and Failure Mode and Effective Analysis (FMEA) it is possible to formulate safety requirements with variables and measurements of the operational model. Besides the benefit is that the safety requirements are simplified. This means that instead of a few very complex safety requirements, the result is a greater number of less complex safety requirements.

The considerations in this paper are focused on the second difficulty by supporting users with expert knowledge and experiences, which are captured in formal specification patterns for safety requirements. The original idea of patterns is to capture recurring solutions, compare [12]. Patterns are meaningful in the case when a user does not need the full expressiveness of the used language to solve the specification problems. This is the situation of formal requirements specification languages like e.g. CTL, CTL*, LTL (Linear Temporal Logic), TLA (Temporal Logic of Actions) or the μ-calculus. All these are variants of Temporal Logic. For most safety requirements, which appear in practice and in literature, relatively simple formalisms are necessary, compare [10].

This paper is focused on the formal specification of requirements in context to safety. For this reason in section 2 an introduction to the characteristics of safety requirements is presented. After these fundamentals in section 3, the idea of our approach to overcome the barriers of the difficulty of formal safety requirements specification is explained. On that basis in section 4 the classification scheme of our approach is introduced. Related work is discussed in section 5. Finally the paper concludes with the most important results and with a discussion about planned future works.
2. Safety Requirements

Our main goal is to simplify the formal specification of safety requirements for software engineers of industrial automation systems. Because of this target it is important first to orient the investigations according to the terminology used in industrial automation technology. Not all possible formal formulas are suitable to express safety requirements. Therefore it is important to consider, what exactly safety means in order to have a foundation for classification, based on suitable formal formulas.

The definition of safety in context to formal languages differs to the terminology of safety in industrial automation technology. The definition in [25] and [20] for a "canonical safety formula" in Temporal Logic is (2) wherein \( p \) is a past formula\(^3\).

\[
"\text{Safety}": \ [\ p \ ] \tag{2}
\]

This means "henceforth \( p \)". In [25] it is explained that a property that can be specified by this "canonical safety formula" is called a safety property. But from the viewpoint of industrial automation technology it cannot be asserted that a property or requirement that can be specified by a special formal formula is always called a safety property resp. requirement. It only can be stated, that a safety requirement, must contain a certain formal formula. Safety in industrial automation technology means: "Safety is freedom from accidents or losses", see [23]. But in [23] it is also stated that there is no such thing as absolute safety. Safety is not an absolute value and therefore safety should be defined as a judgement of the acceptability of the risk for danger. A system is safe if its attendant risks are judged to be acceptable, see [23] and [24]. In this way, safety means absence of danger. Therefore the German DIN VDE 31 000 part 2 defines a safe situation as a situation in which the risk is lower than the maximum acceptable risk, see figure 1, compare [22] and [8].

![Fig. 1.: Definition of safety in (DIN VDE 31 000 part 2)](image.png)

But the "safety" formula, as it is defined in (2), can also be used for reliability requirements in context to industrial automation systems. Reliability means prohibition of failures of an industrial automation system, while safety means prohibition of danger, see [22]. Reliability is the "ability of a system to operate correctly for a specified time. This assumes, that the system is correct at the start of use and only failures can lead to incorrectness", see [14].

In practice a reliability requirement for formal verification is not declared with a special probability. In the same manner safety requirements are not specified with a
declaration of the specific acceptable risk. Besides, the failures in industrial automation systems have their origin in hardware. The goal is, that the software must not have any failures. For these two reasons "henceforth p" can also be used for reliability requirements of software models and not only for safety. That is why "henceforth p" is not always a safety requirement in context to industrial automation technology. But the following is true with regard to the definition in (2): Experience shows that CTL formulas of safety requirements always begin with "always". If the formulations of safety requirements in natural language are analysed, the result is that the expressions in safety must be so strong that something must exist resp. occur, therefore the beginning by "always". A property that may exist is too weak for safety.

Now it still needs to be answered, what safety requirements are in general. Safety requirements are necessary for safety critical systems. While functional requirements are requirements pertaining to all functions that are to be performed by the target system in each mode, safety requirements are requirements about the safe operation of the target system. Safety requirements are to be configured so that if complied with, danger is precluded, see [14].

Safety requirements include information about the safe and unsafe states of the target system and, if possible, the acceptable probabilities for entering an unsafe state, compare [29]. They contain a survey of the possible hazards to people or the environments caused by faults in, or maloperations of the target system.

Therefore not all formulas in formal languages are suitable for safety requirements. If a formula expresses e.g. that a property exists, may exist, may or might be valid any time or occurs eventually, this will be too weak for the formulation of a safety requirement. A requirement in context to safety must always be expressed in such a manner that something must be, i.e. a property must be valid at some certain model states or a required sequence must be executed under some certain preconditions. Safety requirements are always expressed in an imperative manner.

3. Patterns for Formal Specification of Safety Requirements

3.1 The Idea of Formal Specification Patterns

To handle the difficulty 2 introduced in section 1, there exist mainly two kinds of approaches: The first is the use of graphical notations, which visualise the difficult semantics, that has to be specified (STD – Sequence Timing Diagrams, see [30] and [31]; GIL – Graphical Interval Logic, see [7] and [26]; LSC – Life Sequence Charts, see [6]; SPCTL – UML Sequence Diagrams embedded in an extended notation by CTL, see [2] and [4]). The practical benefit of this kind of approaches is obvious but the correct use of these notations still have to be learned and there is still no support by these approaches in learning the difficult semantics.

The second kind of approaches are structured natural languages, compare [18] and [11]. There Temporal Logic part expressions are replaced by part sentences in natural language. To specify a requirement, given part sentences have to be combined together and variables of the operational model have to be inserted.

On the one hand the use of natural language modules is easier to understand the semantics than by Temporal Logic formulas. But on the other hand the correct formal
use of the natural language modules has to be learned and these approaches does not assist in learning the correct use. Often the result of such specifications, are long sentences with artificial sounds. This leads to a rest risk that the artificial sentences could be misunderstood.

Therefore the consequence is to use formal modules and this leads to our approach. In this approach, which is presented in this paper, pre-specified generic safety requirements are used. As far as possible these generic requirements are patterns for complete requirements instead of modules so that the difficult learning process of the right combination of modules and mistakes are avoided. The pre-requisition for this approach is the "hypothesis that only a small fraction of the possible properties that can be specified using logics or regular expressions commonly occur in practice", [10]. Formal specification patterns are used in the following way:

1. The user selects and determines the suitable formal formula in a list with all kinds of common formal specification patterns for safety requirements.
2. In the second step the user has to adopt the pattern to the respective safety requirement in context to the operational model. As a result we get a formal specified safety requirements, which are instances of the patterns in the list.

The list is a catalogue, in which every specification pattern is stated in the formal notations CTL (e.g. for use of the model checker SMV or VIS), LTL (e.g. for use of the model checker SMV or SPIN) and μ-calculus (for the use of the model checker μ-cke). In that way the user is able to choose the formalism of his choice, which he requires for the used formal verification tool. Additionally every pattern is specified in SDCTL, which is easy to handle and for which a transformation into CTL and into the μ-calculus exists. Furthermore every specification pattern is explained in natural language, so that the meaning of the pattern is easier to understand and learnable. Besides, this explanation can be used to know the correct formulation for a safety requirement in natural language. A second user who reads a safety requirements expressed in this way can look up the formal meaning in the catalogue of specification patterns. In this manner this approach can also support the communication in team development.

3.2 Benefits of Patterns for Formal Specification of Safety Requirements

One of the benefits of specification patterns is that they help to formulate correctly safety requirements in formal notation. Furthermore the approach helps to formulate correctly safety requirements in natural language. The main advantage in comparison to other approaches is that the specification patterns transfer expert knowledge. Every pattern has been specified by an expert of formal methods. So if an engineer applies a pattern he will use expert knowledge in specifying a safety requirement in a formal notation. By applying the patterns, this approach supports in learning the application of formal languages for safety requirements specification without needing the help of patterns. Therefore it is an answer to the problem that common software engineers are not experts in logic. The use of the patterns could be a practical education method as an introduction to the practical use of formal specification languages.
Finally the specification patterns represent a recipe book, which contains all common generic safety requirements in formal notations. Therefore it does not only help in the correct formulation of safety requirements it also helps in detect relevant safety requirements. That is because the recipe book can be used as a checklist on which safety requirements could exist in general. Thereby on the basis of such a checklist a user could think about which kind of safety requirements are relevant for the respective system development.

4. Classification of Patterns

4.1. Basic Classification

To support the user to detect the suitable patterns an organisation of the patterns is needed, in which the patterns have been put in order. Such an organisation must show the user the distinct characteristics of the different specification patterns.

In [1] the criteria and the proceedings, which led to the classification, which is presented here, are explained. Because of space limitations in this paper only the main classification of the patterns is presented in this section without the formal notations of the patterns. The patterns in detail are shown on a web site [3].

In our classification scheme based on our experiences all common safety requirements are considered. By the prerequisite that they are derived by using FMEA and FTA, see [2], safety requirements usually are not more complex as they are in our pattern catalogue, presented in [1] and [3].

In figure 2 the first view at the classification scheme is shown. The scheme has to be read from left to right. From left to right the classes are refined by subclasses. In the following the classes are explained.

<table>
<thead>
<tr>
<th>Static Safety Requirements (Invariants)</th>
<th>Safety Requirements with Temporal Dependencies</th>
<th>Safety Requirements about Chronological Succession</th>
<th>Safety Requirements with Explicit Time</th>
<th>Safety Requirements about General Access Guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Static Safety Requirements (Invariants)</td>
<td>A property ( p ) must be &quot;true&quot; in the whole operational model. For example, the safety requirement for a traffic light crossing: &quot;In all situations it is not permitted that the traffic lights of the main road and of the side road display a green signal at the same time.&quot; In the whole operational model both the traffic lights must not display a green signal at the same time.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. Dynamic Safety Requirements
A property $p$ must be "true" in certain model states and in other cases "false". For example, the safety requirement for a railroad crossing: "Only after the train has passed the railroad crossing from then on it is permitted that the gates are opened." In certain model states the gates may be opened but not in the other cases.

2.1. Safety Requirements about General Access Guarantee
These are requirements, which concern to all model states. Therefore they are called "general". But in difference to class 1 the property is not only about the current state, but also about future states. From all model states it must be possible to access to a certain property $p$ or it must be possible to reach $p$. For example, the safety requirement for any safety critical system, which contains an emergency brake: "In all situations it must be possible to actuate the emergency brake." That means from all situations the actuation of the emergency brake must be accessible by reaching a next state of the state space in which the emergency brake is actuated.

2.2. Safety Requirements with Temporal Dependencies
There exists a temporal dependence between propositions. For example, the safety requirement for a control of a pneumatic brake system: "If a defect is detected at a certain valve, the software control system has to be switched off with a certain delay time. Directly after that the redundant pneumatic control has to be switched on." In this safety requirement the temporal dependence exists between the event of the detected defect, the action of switching off the software control and the action of switching on the backup pneumatic control.

Besides for classification it is decisive as to when exactly the required statement must begin and for how long it is valid. Furthermore in automation technology safety requirements are often real-time requirements. These criteria lead to the following subclasses.

2.2.1. Safety Requirements about Chronological Succession
Especially in automation technology there are requirements, in which the exact chronological succession of propositions is important (e.g. in a bus system). In these requirements the beginning and duration of propositions are not dependent on a certain time counter but are dependent on the occurrence of other properties. Class 1 includes the case "for all model states $a \rightarrow b$ must be true". This case possesses an "if then" dependence. If $a$ occurs then $b$ must also be true. But it can be differentiated when exactly $b$ must be valid. The exact temporal dependence between propositions is important. The distinguishing marks are oriented according to how long exactly the predecessor and from when exactly the successor event is permitted to be valid.

2.2.1.1. Safety Requirements about Beginning of Validity
These are requirements about the beginning of the validity of a property, which is in a certain sequential dependence on other properties. For example, the safety requirement for a railroad crossing: "The safeguard of a level crossing is only permitted to be terminated, strictly after the railroad crossing has been completely

---

4 "$a \rightarrow b$" stands for implication between $a$ and $b$, suggesting that $b$ is a logical consequence of $a$. 
vacated if the train had passed.” So the beginning for the termination of the safeguarding is the complete vacation of the railroad crossing.

2.2.1.2. Safety Requirements about Duration of Validity
These are requirements about the duration respectively ending of the validity of a property, which is in a certain sequential dependence on other properties. For example, the safety requirement for a distillation tower: ”The inflow must only be opened until the temperature sensor has relayed the value 400 K.”

2.2.1.3. Safety Requirements about Beginning and Duration of Validity
These are requirements about the beginning and in addition, the duration of the validity, which is in a certain sequential dependence on other properties. For example, the safety requirement for a distillation tower: ”Only after the temperature sensor has relayed the value 350 K, from then on it is permitted that the inflow is opened but only as long as the level of the tank has not reached the minimum value.”

2.2.2. Safety Requirements with Explicit Time
In these requirements the beginning and duration are dependent on certain time duration or certain points in time dependent on a time counter. For example, the safety requirement for a railroad crossing: ”The gates must be in the closed state for 6 seconds before the railroad crossing has the status safeguarded.” In general it can be differentiated between operational models, which are time triggered vs. those which are event triggered. For these cases the formal formulas are different.

In the following subclasses it also can be differentiated between requirements about beginning, duration as well as about beginning and in addition, the duration of validity, compare classes 2.2.1.1 to 2.2.1.3.

2.2.2.1. Safety Requirements for Event Triggered Operational Models
These are requirements, which contain explicit time properties for systems, which are not triggered by time steps. Therefore time in these requirements has to be expressed explicitly. The precondition is that there exists a time counter, which is independent of the operational system.

2.2.2.2. Safety Requirements for Time Triggered Operational Models
These are requirements, which contain explicit time properties for systems which are triggered by time steps. Therefore time properties can be expressed by the specification of certain clock cycles.

4.2. Orthogonal Classifications
All subclasses of the class 2.2 ”Safety Requirement with Temporal Dependencies” can each be distinguished in the following four cases: Requirements which specify a
1. necessary behaviour,
2. permitted or forbidden behaviour,
3. necessary behaviour which is only permitted and
4. behaviour, which must be guaranteed under certain conditions (the possibility or accessibility of a behaviour must be guaranteed if certain temporal preconditions are fulfilled).
The third case has to be considered because in many cases it is not only a simple combination of requirements of the first two classes in formal formulations. In difference to class 2.1 in the fourth case some certain temporal preconditions must be fulfilled to enable a guarantee of accessibility.

For all the classes of ”Dynamic Safety Requirements” we get further subclasses if the interest is not the validity of a single proposition at certain states of the state space but the sequence of several events, actions and/or conditions in the state space. For theses subclasses the use of the language SDCTL seems more suitable and easier than CTL. SDCTL is an integrated language of UML Sequence Diagrams and the formal specification language CTL, compare [2] and [4]. At SDCTL a sequence of events, actions and conditions is specified in SDs. Then the SDs are embedded in a SDCTL-formula. It has to be done this way, that statements are build similar to like it is done for single propositions in class 2. SDCTL may be used instead of CTL formulas because it is possible to transfer the SD part to CTL automatically.

4.3. Further Classes

From the introduced classes and the respective patterns in [1] and [3] further patterns can be derived for following cases:
1. There could be many more propositions as successors in requirements of chronological succession than two but the principle scheme of the patterns is always an analogue to the consideration of only two.
2. Further formal constructs of safety requirements result from a combination of the temporal logic statements of the several classes by connectives of Propositional Logic: AND, OR, XOR, NOT and ¬.

5. Example of the Application of Safety Patterns

The application of the safety patterns shall be demonstrated by a short example. For the safety requirement ”The gates must not be opened before the train has passed the railroad crossing” for a railroad crossing control the suitable formal specification in CTL would have to be detected. For that purpose first the safety requirement has to assigned to the correct classes of the classification levels step by step:
1. Only in certain states of the state space it is permitted that the gates are opened. Therefore the requirement belongs to the class ”Dynamic Safety Requirements”.
2. There exists a temporal dependence between this, ...
   a. that the train passes the railroad crossing and
   b. that the gates are opened. Therefore the safety requirement belongs to the class ”Safety Requirements with Temporal Dependencies”.
3. There is no temporal statement in dependence on a system clock. For that reason it is a ”Safety Requirement about Chronological Succession”.
4. It is a safety requirement of when exactly the gates may be opened. Therefore the safety requirement has to be classified as ”Safety Requirements about Beginning of Validity”.

5. Only after the train has passed the railroad crossing it is **permitted** that gates are opened. Otherwise the opening is **forbidden**. That is why a behaviour has to be specified, “which is only permitted”.

6. Only one action is the subject namely the opening of the gates and not a sequence of actions, events and/or conditions. For this reason the safety requirement has to be assigned to the class "Safety Requirement about the validity of one proposition”.

By this classification with help of the safety patterns catalogue in [3] the following formal formula with the appropriate explanation can be found:

**Formal formula in CTL:** \( A((\neg p) \land (q \land \neg p)) \) \hspace{1cm} (3)

**Explanation:** *Only after an event p has occurred, from then on it is permitted that an action q is executed.*

With help of this safety pattern a safety requirement in context of the operational model can be specified:

\( A((\neg opening) \land (train\_crossed \land \neg opening)) \) \hspace{1cm} (4)

"Only after the train has passed the railroad crossing, from then on it is permitted that the gates are being opened."

### 6. Related Works

The most popular classification of requirements is in [21]. It contains the distinction of "nothing bad will ever happen" vs. "something good will eventually happen". But this is too coarse to be of practical use for requirements specification in meaning of specification patterns.

A further work, which contains a finer classification, is from Manna and Pnueli (s. [25]). But the approaches differ mainly in two points. First they proceed from the assumption that very little of the general theory of Temporal Logic is required to handle the major and most common requirements of concurrent programs. Therefore their categories are much broader than our classification. But our experiences with software development for industrial automation systems and the experiences shown in [10] are, many more kinds of formal patterns have to be considered.

The second difference is the terminology used and this is also a difference from our approach to [21]. They use safety in another meaning, that also means reliability and not only safety in context to industrial automation systems, compare section 2. But our approach agrees in that point that a safety requirement always begins with the statement "always”.

Closer to our approach is [9] and [10] of Dwyer, Avrunin and Corbett. They have developed a pattern system, which is concerned with "the translation of particular aspects of requirements into the formal specification is suitable for use with finite-state verification tools”, [10]. The agreement with our approach is the conviction about the benefit of formal specification patterns. They have collected many experiences with extensive practical studies.
The main difference of their approach is, that they do not restrict the practical use of formal verification to context to safety. For this reason theirs considered kinds of specification patterns are not restricted to safety requirements. Therefore the kinds of patterns are more voluminous. If the interests of a user in formal specification were only in context to safety, their pattern system would be more difficult to use, because there are many patterns, which are not relevant to safety requirements in general. They organised the patterns in a hierarchy based on their semantics while in our approach the following criteria has been mainly considered for classification, compare [1]: The classification has been mainly decided by considering the terminology of industrial automation systems, by different cases of formal formulas and by different cases in formulation of safety requirements in the natural language. Especially only these cases were considered which are meaningful in context to the terminology of safety. Therefore our patterns are "safety patterns". At the development of the classification of our approach we placed value on practical relevance of the safety patterns for industrial automation systems. A result is e.g. that we consider requirements with explicit time in own classes. Another example is that we distinguish the exact succession of propositions in detail, this way whether there may be an overlap between the occurrence of the successor and the predecessor. This is e.g. important for bus systems. Finally we consider explicit extra classes for sequences of events, actions and conditions. All these differences of our work compared to [9] and [10], lead to another order of our classification scheme.

7. Summary and Outlook

Formal verification by the use of model checking is more and more important in software development of safety critical systems. Thereby safety requirements have to be expressed in a formal specification language. But the practical use of this kind of specification language is difficult for software engineers who are not experts in logic.

A way to overcome these difficulties is the use of generic safety requirements in a formal notation. That means the user specifies the safety requirement in a formal notation with the help of a catalogue of safety patterns. Such a catalogue provides information on formalisation, transfers expert knowledge and could help in learning the use of formal languages for safety requirements specification. Besides the safety patterns catalogue could be used as a checklist on which safety requirements could exist in general and therefore supports to detect relevant safety requirements.

We do not claim that the introduced catalogue is complete now. But based on our today’s experiences all kinds of safety requirements of software models of industrial automation systems are included in the safety patterns presented in [3].

We are still evaluating the safety patterns and are collecting further practical experience with the help of case studies especially in the development of software in the automotive and railway control areas. We are grateful for every constructive contribution for our safety patterns catalogue especially by experienced users of formal methods and software engineers of safety critical systems.

For practical usability of the explanations of the patterns in natural language there will be developed variants of formulations. The explanations shall be usable for correct formulation of a safety requirement in natural language. By the offering of
variants, it will be possible to select a formulation, which sounds most possible natural in context to the respectively application.

Besides, it will be necessary to investigate the different claims concerning the characteristics of safety requirements in the several levels of the development process. Further pattern classes could be found specific to the several development levels.

References

8. DIN VDE 31000 Teil 2, Begriffe der Sicherheitstechnik: Grundbegriffe.
Formal Support for Fault Modelling and Analysis

Tadeusz Cichocki and Janusz Górski

1 Adtranz Zwus, Modelarska 12, 40-142 Katowice, Poland, tadeusz.cichocki@pl.transport.bombardier.com
2 Technical University of Gdańsk, Narutowicza 11/12, 80-952 Gdańsk, Poland, jango@pg.gda.pl

Abstract. The paper presents how CSP and the associated tool FDR are used to support FMEA of a software intensive system. The paper explains the basic steps of our approach (formal specification, systematic fault identification, fault injection experiments and follow-up) and gives some results related to the application of this method to the industrial case study, a railway signalling system that is presently under development.

1 Introduction

Success of a safety-critical system development project depends to large extent on the designer’s ability to include in the design adequate defences against a justifiably complete class of faults and to prove the correctness of the design against the normative rules and requirements. This objective is supported by the FMEA (Failure Mode and Effect Analysis) method. The method recognises the system components structure and admits that components’ failures can affect higher level (system) properties. FMEA assumes that component faults are identified in a systematic way, investigates the effects of those faults on the system properties and then, if necessary, the findings are taken into account in the following design decisions. It has been argued [1], [3], [4] that, for software intensive systems, FMEA can benefit if it is supported by a formal method – this way we can increase precision and remove ambiguities from the analyses.

In [1] a general context of our research is presented. There and in this paper we investigate the application of the CSP [5] notation as a formal method supporting the FMEA process. Our presentation refers to the Line Block System industrial case study that is presently under development. The present paper shows how we use CSP and the associated tool FDR [2] to identify component faults and to analyse the consequences those faults can have in the higher-level system components. Our method comprises the following main steps:

- formal specification of the system and its components,
- systematic fault identification referring to the formal specification,
- fault consequences analysis through fault injection experiments,
- follow up: fault acceptance or specification redesign.

The steps are explained in the subsequent sections of the paper.
2 The Line Block System Case Study

Line Block System (LBS) is a railway signaling system presently under development at Adtranz Zwus in Katowice, Poland. In [1] it has been shown how we used object-oriented approach to develop a model of this system. The model is hierarchical and represents the system architecture on subsequent levels of increasing detail. Fig.1 presents the collaboration diagram [6] of LBS. It shows the objects of the system and the channels of their co-operation. The BLOCK object supervises a single sector of the rail track. It communicates with train detectors (det) and semaphores (sem). It also communicates with similar blocks on its left and right (represented by the UNIT_next and UNIT_before objects).

In Fig.1 we also show the internal structure of BLOCK (included in the dotted rectangle). This is a more refined level of the model where the higher level object is represented in terms of its components. BLOCK is shown as being composed of LBC, DETECTING_DEV and SIGNALLING_DEV. Input and output channels of BLOCK are inputs and outputs of its component objects. In addition, the components can exchange signals that are not visible outside BLOCK (are internal to BLOCK). Test, confirm, dd and sd are examples of those.

![Fig. 1. Line Block System collaboration diagram.](image)

The advantage of object-orientation is that the model closely follows the actual structure of the system and therefore is easily understood by system designers and implementors. Hierarchical modelling supports the distinction between the design and implementation oriented views of the system. This provides a framework within which we can analyse possible influences the lower level components can have on their higher level “containers”, which is the essence of FMEA.

3 Formal Specification and Verification

To provide for precision and unambiguity we apply formal specifications to our models. Our choice was CSP [5] as this notation provides for specification of objects interacting through communication channels. The CSP objects (processes) interact with each other and the environment by means of communication events
(instantaneous atomic actions). The CSP expressions describe patterns of event causality and the way the cooperating processes synchronize. The synchronization is achieved on specific events and may involve data exchange between processes.

A CSP process is observed by the traces (the sequences of events) the process can engage in. For a process $P$, we define $\text{alpha}(P)$ as the set of all events $P$ is able to synchronize on and $\text{traces}(P)$ as the set of all possible finite traces of $P$. A failure of a process is a (finite) trace together with a refusal set which is a set of events that the process might refuse to engage in after performing the trace (note that if this set equals $\text{alpha}(P)$ then $P$ deadlocks – refuses to engage in any event). For a process $P$, $\text{failures}(P)$ denotes the set of all failures of $P$. The divergences of a process $P$, denoted $\text{divergences}(P)$, is a set of traces after which $P$ may diverge, i.e. perform an infinite sequence of internal (invisible) events.

Three models of process behaviour are considered: traces (T), failures (F) and failures-divergences (FD). In the traces model, $P$ is characterised by $\text{traces}(P)$ and, by definition, the traces refinement relation between two processes $P$ and $Q$ (denoted $P \mathrel{T=} Q$) holds iff $\text{alpha}(P) = \text{alpha}(Q)$ and $\text{traces}(Q) \subseteq \text{traces}(P)$.

In the failures model, $P$ is characterised by the $\text{failures}(P)$ set and the failures refinement relation between two processes, $P$ and $Q$ (denoted $P \mathrel{F=} Q$) holds iff $\text{alpha}(P) = \text{alpha}(Q)$ and $\text{failures}(Q) \subseteq \text{failures}(P)$.

In the failures-divergences model, $P$ is characterised by the $\text{failures}(P)$ and $\text{divergences}(P)$ pair of sets and the failures-divergences refinement relation between two processes, $P$ and $Q$ (denoted $P \mathrel{FD=} Q$) holds iff $\text{alpha}(P) = \text{alpha}(Q)$, $\text{failures}(Q) \subseteq \text{failures}(P)$ and $\text{divergences}(Q) \subseteq \text{divergences}(P)$.

Note that for divergence free processes, the relations $\mathrel{FD=}$ and $\mathrel{F=} are equivalent.

To specify our processes we use the CSP dialect supported by the FDR (Failures Divergence Refinement) [2] tool. This gives rise for subsequent application of FDR as an analytical tool. Using FDR we can verify various properties of the specifications, including:

- deadlock freedom – a process never enters a state where there is no possibility of continuation (execution of the next event),
- divergence freedom – a process never enters a state where an infinite sequence of internal events is possible without any external event occurrence,
- semantic relations of processes – verification of the traces refinement, failures refinement and failures-divergences refinement relations CSP between processes.

Below we enclose formal specifications of BLOCK and its components. To save space we omit the specification of communication channels and concentrate on the behaviours of the objects. The specification of BLOCK slightly differs from this presented in [1]. Here we additionally distinguish a safe state of BLOCK (BLOCK_safe_state) and in the mission oriented part of BLOCK specification (BLOCK_interlocking) we explicitly handle a case of possible detector malfunction (the detectors.INO clause). In the specification we refer to five standard signals defined for a line-block system: S1 – S5 that are displayed on semaphores.
-- BLOCK

BLOCK = BLOCK_interlocking |~| BLOCK_safe_state
-- The symbol |~| denotes the internal choice operator.
-- Behaviour of BLOCK in its two basic states is shown.

BLOCK_interlocking = detectors.IN -> BLOCK_occupied
   [[] detectors.OUT -> BLOCK_not_occupied
   [[] detectors.INO -> BLOCK_occupied
-- The symbol [] denotes the external choice operator

BLOCK_occupied = sem.S1 -> signal_before.S1 -> BLOCK
   |~| sem.S0 -> signal_before.S1 -> BLOCK
-- Possible failure of the synchronization on sem.S1;
-- S1 is 'red' (stop) signal and S0 is a dark one.

BLOCK_not_occupied =
   signal_next.S0 -> sem.S5 -> signal_before.S5 -> BLOCK
   [[] signal_next.S1 -> sem.S5 -> signal_before.S5 -> BLOCK
   [[] signal_next.S2 -> sem.S2 -> signal_before.S2 -> BLOCK
   [[] signal_next.S3 -> sem.S2 -> signal_before.S2 -> BLOCK
   [[] signal_next.S4 -> sem.S3 -> signal_before.S3 -> BLOCK
   [[] signal_next.S5 -> sem.S3 -> signal_before.S3 -> BLOCK
-- Implementation of the signalling interlocking rules

BLOCK_safe_state = sem.S1 -> signal_before.S6 -> STOP
   |~| sem.S0 -> signal_before.S6 -> STOP
-- The BLOCK's fail safe state.

The above specification gives all possible traces of events that can be observed at
the BLOCK interface. The specification can be submitted to the FDR tool in order to
verify some of its properties. Examples of assertions to be validated are given below:
assert BLOCK : [deadlock free]
assert BLOCK : [divergence free]

Positive validation of the above assertions means that the BLOCK specification is
deadlock and divergence free.

At the more refined level the model explains the internal structure of BLOCK (see
the dotted rectangle in Fig.1). Again, we use CSP to specify the BLOCK components.
The specifications follow (again, the declarations of channels are omitted).

-- LBC

-- Line block controller
LBC = test.interlocking -> LBC_interlocking
   |~| test.safe_state -> LBC_safe_state

LBC_interlocking =
   dd.IN -> LBC_occupied
   [[] dd.OUT -> LBC_not_occupied
   [[] dd.INO -> LBC_occupied

LBC_safe_state = sd.S6 ->
   (confirm.S1 -> signal_before.S6 -> STOP
    [[] confirm.S0 -> signal_before.S6 -> STOP)

LBC_occupied = sd.S1 ->
   (confirm.S1 -> signal_before.S1 -> LBC
[] confirm.S0 -> signal_before.S1 -> LBC)

LBC_not_occupied =
signal_next.S0 -> sd.S5 -> confirm.S5 ->
signal_before.S5 -> LBC
[] signal_next.S1 -> sd.S5 -> confirm.S5 -> signal_before.S5 -> LBC
[] signal_next.S2 -> sd.S2 -> confirm.S2 -> signal_before.S2 -> LBC
[] signal_next.S3 -> sd.S2 -> confirm.S2 -> signal_before.S2 -> LBC
[] signal_next.S4 -> sd.S3 -> confirm.S3 -> signal_before.S3 -> LBC
[] signal_next.S5 -> sd.S3 -> confirm.S3 -> signal_before.S3 -> LBC

-- DETECTING_DEV
-- Device detecting presence of a train in the BLOCK
DETECTING_DEV =
test.interlocking -> DETECTING_DEV_interlocking
[] test.safe_state -> STOP

DETECTING_DEV_interlocking =
detectors.IN -> dd.IN -> DETECTING_DEV
[] detectors.OUT -> dd.OUT -> DETECTING_DEV
[] detectors.INO -> dd.INO -> DETECTING_DEV

-- SIGNALLING_DEV
-- Device signalling states of the BLOCK to trains.
SIGNALLING_DEV =
sd.S1 -> SIGNALLING_occupied
[] sd.S2 -> SIGNALLING_not_occupied(S2)
[] sd.S3 -> SIGNALLING_not_occupied(S3)
[] sd.S4 -> SIGNALLING_not_occupied(S4)
[] sd.S5 -> SIGNALLING_not_occupied(S5)
[] sd.S6 -> SIGNALLING_safe_state
-- S6 is a special purpose signal used in LBS.
SIGNALLING_safe_state =
sem.S1 -> confirm.S1 -> STOP
|~| sem.S0 -> confirm.S0 -> STOP
-- Failure of the synchronization on sem.S1
SIGNALLING_occupied =
sem.S1 -> confirm.S1 -> SIGNALLING_DEV
|~| sem.S0 -> confirm.S0 -> SIGNALLING_DEV
-- failure of the synchronization on sem.S1
SIGNALLING_not_occupied(spar) =
sem.spar -> confirm.spar -> SIGNALLING_DEV

Having specified the components we formally declare that together they form
BLOCK_IMP – the implementation of BLOCK.

-- BLOCK_IMP
BLOCK_IMP = DETECTING_DEV
[]|{|dd,test|}| (LBC []|{|sd,confirm|}|) SIGNALLING_DEV)
A standard step to be performed now is to compare the specifications of BLOCK and BLOCK_IMP in order to verify if they are consistent. This can easily be done with the help of FDR. The assertions to be validated are given in Table 1 below.

**Table 1. Verification conditions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Refinement condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>BLOCK [T= BLOCK_IMP \ {{dd, test, sd, confirm}}]</td>
</tr>
<tr>
<td>R2</td>
<td>BLOCK [FD= BLOCK_IMP \ {{dd, test, sd, confirm}}]</td>
</tr>
<tr>
<td>R3</td>
<td>BLOCK_IMP \ {{dd, test, sd, confirm}} [T= BLOCK]</td>
</tr>
<tr>
<td>R4</td>
<td>BLOCK_IMP \ {{dd, test, sd, confirm}} [FD= BLOCK]</td>
</tr>
</tbody>
</table>

The tool reports the positive result by showing (according to the FDR convention) green tick ![green tick](image)before each of the assertions.

### 4 Systematic Fault Identification

Let us consider a CSP specification of an object A. Consider all possible deviations of the interface events from their specifications preventing the object’s synchronisation with its environment. The deviations include modification of the external events set and modifications of the channel types. Each deviation is then assessed concerning the likelihood of its occurrence in a real system. Those deviations that are positively validated are then included in the Fault Table of the object A. We call them **syntactic faults**.

Another deviations that we consider are those that affect the causality pattern of the object behaviour. We consider possible event scenarios that are inconsistent with the object’s internal state (are not implied by the state), but result in synchronisation between A and its environment. Such deviations include events which inconsistency can not be detected by the co-operating components. We consider all possibilities of such events and then assess the likelihood of their occurrence in the real system. Those that are positively validated are included in the Fault Table as well. We call them **semantic faults**.

The analysis of the BLOCK object proceeded as follows.

Possible syntactic and semantic faults of the component objects were generated from their specifications. The analysis covered all possible violations for each channel. The faults were subjected to the validation argumentation (to assess the likelihood of their occurrence) and then documented in the corresponding Fault Table. The following tables, Table 2, Table 3 and Table 4, contain examples of the faults of the components of BLOCK.

**Table 2. Fault Table of DETECTING_DEV (extract)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Fault description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal synchronization</td>
</tr>
<tr>
<td>DV1</td>
<td>detectors.IN -&gt; dd.IN</td>
</tr>
<tr>
<td>DV2</td>
<td>detectors.INO -&gt; dd.INO</td>
</tr>
<tr>
<td>DV3</td>
<td>detectors.INO -&gt; dd.INO</td>
</tr>
<tr>
<td>DV4</td>
<td>detectors.OUT -&gt; dd.OUT</td>
</tr>
</tbody>
</table>
Table 3. Fault Table of SIGNALLING_DEV (extract)

<table>
<thead>
<tr>
<th>Name</th>
<th>Normal causality</th>
<th>Faulty causality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV1</td>
<td>Sd.SB4 -&gt; SIGNALLING_not_occupied(S4)</td>
<td>sd.SB4 -&gt; SIGNALLING_not_occupied(S3)</td>
</tr>
<tr>
<td>SV2</td>
<td>Sd.SB5 -&gt; SIGNALLING_not_occupied(S5)</td>
<td>sd.SB5 -&gt; SIGNALLING_not_occupied(S4)</td>
</tr>
</tbody>
</table>

Table 4. Fault Table of LBC (extract)

<table>
<thead>
<tr>
<th>Name</th>
<th>Normal synchronization or causality</th>
<th>Faulty synchronization or causality</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV1</td>
<td>dd.IN</td>
<td>Declaration of dd channel type extension by INOX and simulation of dd.INOX</td>
</tr>
<tr>
<td>LV2</td>
<td>Signal_next.S5 -&gt; sd.SB3</td>
<td>signal_next.S5 -&gt; sd.SB4</td>
</tr>
</tbody>
</table>

5 Fault Injection Experiments

Each fault included in the Fault Tables is then subjected to what we call a fault injection experiment. Such experiment includes two steps: (1) fault injection and (2) fault consequences analysis.

The fault injection step involves introducing changes to the specification. For syntactic faults it may require changes in the declarations of channel types and some redesign of the component interfaces.

The fault consequence analysis step is performed with the support of the FDR tool. The aim of the analysis is to verify if (and how) a given fault can violate the specification at the higher level (the specification of BLOCK, in our case study). The list of verification conditions for BLOCK is given in Table 1.

The results of the fault injection experiments for the faults of the Tables 2, 3 and 4 are documented in Tables 5, 6 and 7, respectively. The red cross and dot mark (the convention of FDR) denotes that the check has been completed and (at least) one counter-example for the condition in question has been found. The dot • shows that the counter-example is available through the debug option of FDR.

Each fault injection experiment that is not positively validated by the FDR run is then analysed to find out the nature of the detected inconsistency. Of great help here are the counterexamples provided by the tool as they help to identify event scenarios that led to failures. For example, for the LV2 fault of Table 7, the faulty implementation of BLOCK_IMP performs the following sequence of external events (as shown by a trace for R1 and R2 in Table 7): detectors.OUT -> signal_next.S5 -> sem.S1, whereas the allowed sequence for BLOCK is as follows (shown by the trace for R3 and R4 in Table 7): detectors.OUT -> signal_next.S5 -> sem.S3. The analysis of such case leads, in general, to the following decisions:

- Acceptance: we accept the (negative) consequences of the fault. However a message is passed to the designers to increase efforts towards lowering the likelihood of the fault occurrence (e.g. by choosing more reliable technologies),
- Redesign: the present design of the system is changed in order to eliminate the negative consequences of possible occurrence of the fault (for instance, fault detection mechanism and safe state enforcement).

**Table 5. The results for the DETECTING_DEV faults**

<table>
<thead>
<tr>
<th>Fault name</th>
<th>The result of FDR check</th>
</tr>
</thead>
<tbody>
<tr>
<td>DV1</td>
<td>R1: ★★ and R2: ★★</td>
</tr>
<tr>
<td></td>
<td>BLOCK_IMP performs:</td>
</tr>
<tr>
<td></td>
<td>_tau</td>
</tr>
<tr>
<td></td>
<td>test.interlocking</td>
</tr>
<tr>
<td></td>
<td>detectors.IN</td>
</tr>
<tr>
<td></td>
<td>dd.OUT</td>
</tr>
<tr>
<td></td>
<td>signal_next.S2</td>
</tr>
<tr>
<td></td>
<td>R3: ★★ and R4: ★★</td>
</tr>
<tr>
<td></td>
<td>BLOCK performs:</td>
</tr>
<tr>
<td></td>
<td>_tau</td>
</tr>
<tr>
<td></td>
<td>detectors.IN</td>
</tr>
<tr>
<td></td>
<td>_tau</td>
</tr>
<tr>
<td></td>
<td>sem.S1</td>
</tr>
</tbody>
</table>

| DV2        | R1: ★★ and R2: ★★       |
|            | BLOCK_IMP performs:     |
|            | _tau                    |
|            | test.interlocking       |
|            | detectors.NO           |
|            | dd.OUT                  |
|            | signal_next.S2          |
|            | R3: ★★ and R4: ★★       |
|            | BLOCK performs:         |
|            | _tau                    |
|            | detectors.NO            |
|            | _tau                    |
|            | sem.S1                  |

| DV3        | R1: ★★ and R2: ★★       |
|            | R3: ★★ and R4: ★★       |

| DV4        | R1: ★★ and R2: ★★       |
|            | BLOCK_IMP performs:     |
|            | _tau                    |
|            | test.interlocking       |
|            | detectors.OUT           |
|            | dd.IN                   |
|            | sd.SB1                  |
|            | _tau                    |
|            | sem.S1                  |
|            | R3: ★★ and R4: ★★       |
|            | BLOCK performs:         |
|            | _tau                    |
|            | detectors.OUT           |
|            | signal_next.S2          |

**Table 6. The results for the SIGNALLING_DEV faults**

<table>
<thead>
<tr>
<th>Fault name</th>
<th>The result of FDR check</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV1</td>
<td>R1: ★★ and R2: ★★</td>
</tr>
<tr>
<td></td>
<td>R3: ★★ and R4: ★★</td>
</tr>
</tbody>
</table>

| SV2        | R1: ★★ and R2: ★★       |
|            | R3: ★★ and R4: ★★       |
|            | BLOCK_IMP performs:     |
|            | _tau                    |
|            | test.interlocking       |
|            | detectors.OUT           |
|            | dd.OUT                  |
|            | signal_next.S0          |
|            | sd.SB5                  |
|            | sem.S4                  |
|            | BLOCK performs:         |
|            | _tau                    |
|            | detectors.OUT           |
|            | signal_next.S0          |
|            | sem.S5                  |
Table 7. The results for the LBC faults.

<table>
<thead>
<tr>
<th>Fault name</th>
<th>The result of FDR check</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV1</td>
<td>R1: ××× and R2: ×××</td>
</tr>
<tr>
<td>LV2</td>
<td>R1: ××× and R2: ×××</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLOCK_IMP performs:</th>
<th>BLOCK performs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>_tau test.interlocking detectors.OUT dd.OUT signal_next.S5 sd.SB4 sem.S1</td>
<td>_tau detectors.OUT signal_next.S5 sem.S3</td>
</tr>
</tbody>
</table>

The analyses were performed using FDR ver. 2.66 running under the Red Hat LINUX operating system on a PC with INTEL Pentium III 600 MHz processor. The full specification of BLOCK comprised 35 lines of CSP/FDR code. The total size of DETECTING_DEV, SIGNALLING_DEV and LBC comprised 60 lines of CSP/FDR code. The total processing time used for the analyses was 12 hours (13 minutes for one fault). The total numbers of faults considered for the component objects are given in Table 8.

Table 8. The numbers of faults considered during analyses

<table>
<thead>
<tr>
<th>Component name</th>
<th>Number of faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBC</td>
<td>21</td>
</tr>
<tr>
<td>DETECTING_DEV</td>
<td>10</td>
</tr>
<tr>
<td>SIGNALLING_DEV</td>
<td>25</td>
</tr>
</tbody>
</table>

6 Conclusions

The potential of formal methods to support safety analysis of software intensive systems has been well recognised [3], [4]. However, their industrial application still faces several difficulties. One of those is the complexity of formal analysis which, if not supported by powerful tools, quickly goes out of control. The advent of matured tools that can be used by engineers opens a new prospect for application of formal methods in the safety industry. This however requires investigation of possible usage patterns and finding ways the formal methods can support the techniques and methods widely applied by safety engineers.

In the paper we presented a case study of using CSP and the associated FDR tool to support FMEA of a safety related system. The tool was used to support fault injections into specifications and analysis of their consequences. FMEA seems to be well suited to be supported by a formal method because it assumes hierarchical decomposition of the system. Thanks to that the scope of formal analysis can be restricted and focuses on investigation of the relationship between the adjacent layers in the hierarchy. Consequently, the complexity of the formal analysis is limited even if the system under consideration is relatively large.
The models of our approach are developed top-down, starting from the system requirements and going down to the components that are implemented in software or hardware. The analysis goes bottom-up. We identify possible faults and then analyse their consequences in the higher levels of the system structure. In order to be able to pre-select faults (in order to focus only on those that really matter) we have to assess the candidate faults concerning the likelihood of their occurrence. This is achieved by referring to the knowledge that comes from outside of the formal framework (e.g. the component failure profiles, assessment of the technologies used to implement a component etc.).

The method is based on specifications and does not help projects with missing specifications. However, the process of building formal specifications can help in early identification of omissions and inconsistencies in specifications.

The documentation of the process of the formal FMEA can be (and it is) used as a crucial part of the safety case argumentation. It supports explicit enumeration of faults and the visibility of the following decisions (being it fault acceptance or system redesign). The help of the tool in finding an example fault consequence is also appreciated.

References

Project Experience with IEC 61508 and Its Consequences

Rainer Faller
exida.com L.L.C., Wildenholzener Strasse 26
D-81671 München, Germany
Rainer.Faller@exida.com

Abstract. The paper reports on the experience made with the IEC 61508 implementation in recent projects of European and North American system vendors and Japanese equipment vendors. As an answer to the identified problems, the paper describes a knowledge tool to ease a formalized verification process and proposes a combination of software verification methods to address the particular issues with pre-existing software for use in programmable electronic safety systems.

1 Introduction

In the past two decades only a small group of system vendors in the nuclear, avionics, medical, railroad and process industry got in contact with Functional Safety of computerized systems. Now within the relatively short period of three years, the user requirements sections of many biddings require engineering contractors and system suppliers world-wide to comply with the Functional Safety requirements of the international standard IEC 61508.

2 Success, Strength and Weaknesses of IEC 61508

Even if IEC 61508 appears to many people as a totally new set of requirements, the standard is just one in a long chain of standards on Functional Safety of computerized systems and software. New is its positioning as an International Basic Safety Publication by which it stands out of a particular industry sector as other standards for the nuclear and avionics industry do.

IEC 61508 also stands out in its system approach addressing the complete safety installation from sensor to actuator with its technical as well as management issues. This system approach and the world-wide publicity, make Buyers of Programmable Electronic Systems (PES) and Authorities see it as a major reference to reduce their uncertainty on complex systems in their safety applications. The respect IEC 61508 receives at Authorities also drives the hope of decision makers in the plant operator community as well as with the equipment vendors that it will replace national regulations within a few years. This hope is supported by its recent ratification as European Norm EN 61508.
This success is not something one could have expected, considering the volume of the standard and its language both making it a hard reading for people unfamiliar with the subject of Functional Safety.

2.1 Strengths

IEC 61508 introduces a rigorous requirements driven development approach for Functional Safety of applications and equipment alike. Again this is nothing new but it took time to implement such a rigorous approach. As the different projects have now been executed at operating companies and equipment vendors, one can see the benefit this approach provides:

**Operating companies:** More thorough risk analysis leads to reduced costs. It has been reported by SHELL [Wiegerinck 1999] that more than 70% of their Safety Functions in process applications are less than SIL3. Thus many were over-engineered in the past. On the other hand the experienced based safety instrumentation concepts led to a small percentage of Safety Functions which were classified too low.

**User – Vendor relation:** More thorough risk analysis also leads to more precise Specifications of safety functions, timing and safety integrity requirements. This makes it easier for vendors to understand the problem and propose adequate solutions.

**Vendors:** First examples from the development of PES show that the Safety Lifecycle with its rigorous requirements driven development approach leads to hardware and software projects being on time and meeting user expectations more accurately. The author was happy to participate in a project where on the same platform a safety system and a standard system was developed in the same time frame. The development team of the safety system could finally proudly claim that they delivered on time whereas the standard system was months overdue. The safety system development also met the specifications more accurately such that the customer later considered to use safety system modules instead of standard system modules at least for an interim period. It is admitted by now that a rigorous requirements driven development approach enhances accountability of software project schedules.

Due to the strong technical influence of well proven German safety concepts there is a strong emphasis on random Hardware fault investigations in Europe. An example is the standard EN 954-1 for safety-related parts of machine control systems. Whereas this was justified in the past by unreliable electronic components and manufacturing techniques, IEC 61508 puts it in balance with other factors as the Common Cause by introducing probabilistic evaluation. One can demonstrate [Faller 2001] that the Probability of Failure on Demand of a redundant system configuration
is better than a single channel system only if its Common Cause factor $\beta$ is better than:

$$\beta < \frac{(1 - \text{SFF}_{100\text{ID}})}{(1 - \text{SFF}_{100\text{ND}})}$$

SFF stands for Safe Failure Fraction, see tables 2 and 3 of IEC 61508-2;
1oo1D stands for single channel architecture with diagnostics;
1ooND stands for redundant architecture with diagnostics where 1 out of N channels is sufficient to perform the safety function;

An example from the Architectural Constraint table for SIL 3 might help to understand:

$$\text{SFF}_{100\text{ID}} \geq 99\%$$
$$\text{SFF}_{100\text{ND}} \geq 60\%$$
$$\beta < 2.5\%$$

Such low Common Cause factors are not easy to achieve for a homogenous redundant systems, it shows the crucial importance of the Common Cause investigation and the Safe Failure Fraction (diagnostic coverage) in a single channel of the redundant system configuration.

### 2.2 Weaknesses

The publicity of IEC 61508 makes people forget that Application Standards and European Harmonized Standards have absolute dominance over any generic standard or Basis Safety Publication. This leads frequently to situations where programmable electronic systems (PES) were designed and implemented following IEC 61508 but the safety assessor for the application or appliance refuses the approval as, e.g., specific requirements of European industry sector standards like EN 954-1 are not literally met.

As said earlier, IEC 61508 and derived standards are voluminous and quite difficult to read and interpret. Many requirements are not allocated to a certain range of Safety Integrity Levels or to the complexity of the design. This make it difficult to tailor for smaller projects and makes Management of Functional Safety to be (too) expensive for Small and Medium Enterprises upfront. A way to mitigate this weakness will be discussed later in this document.

The possible ambiguity in the interpretation encourages many to use the standard as a toolbox where they take out and require or implement what they understand or like. Whereas in North America, users seem to be mainly concerned about the hardware safety integrity (dangerous failure rates and safe failure fraction), in Europe many users seem to be more concerned about the software safety integrity. The standard also leaves (too) much room for interpretation. In Europe and Germany, most experts interpret the architectural Hardware constraints and diagnostic requirements of part 2 of the standard different than American experts do, leading to situations described in the above paragraph. European industry sector standards such as EN 954-1 and EN 298 do not accept systems where the failure mode of a single component could lead to an unsafe state whereas IEC 61508 does.

Another area where the probabilistic approach of the standard leads to a huge difference in requirements is on pre-existing software and products in low demand mode versus high demand mode application. For the same software the acceptance criteria for the quantitative evaluation of collected and accepted operating data or
statistical tests come out dramatically different. The required number of successfully treated demands for low demand mode of operation can be calculated for a given Probability of Failure on Demand (PFD) and Confidence Level (C) to:
\[ n \geq - \ln(1-C) / \text{PFD} \]

The required number of successful operating hours for high demand or continuous mode of operation can be calculated for a given Probability of dangerous Failure per hour (PdF) and Confidence Level (C) to:
\[ \text{hours} \geq - \ln(1-C) / \text{PdF} \]

Table 1 shows the number of demands resp. operating hours to be executed without showing errors.

<table>
<thead>
<tr>
<th></th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IEC 61508-2</strong></td>
<td>at least 1 year different applications C = 95%</td>
<td>at least 1 year different applications C = 95%</td>
<td>at least 1 year different applications C = 99%</td>
</tr>
<tr>
<td><strong>IEC 61508-7</strong></td>
<td>(Table B.6)</td>
<td>(Table D.1)</td>
<td></td>
</tr>
<tr>
<td><strong>Low demand mode of operation</strong></td>
<td>PFD $\geq 10^{-2}$ results in</td>
<td>PFD $\geq 10^{-3}$ results in</td>
<td>PFD $\geq 10^{-4}$ results in</td>
</tr>
<tr>
<td><strong>Required demands executed without errors</strong></td>
<td>$n \geq 300$</td>
<td>$n \geq 3,000$</td>
<td>$n \geq 46,000$</td>
</tr>
<tr>
<td><strong>High demand or continuous mode of operation</strong></td>
<td>PdF $\geq 10^{-6}$ results in</td>
<td>PdF $\geq 10^{-7}$ results in</td>
<td>PdF $\geq 10^{-8}$ results in</td>
</tr>
<tr>
<td><strong>Required number of operating hours executed without errors</strong></td>
<td>hours $\geq 3\times10^6$ years $\geq 342$</td>
<td>hours $\geq 3\times10^7$ years $\geq 3420$</td>
<td>hours $\geq 4.6\times10^8$ years $\geq 52,6\times10^3$</td>
</tr>
</tbody>
</table>

It is obvious from these figures, that it is very difficult to demonstrate proven-in-use for systems which shall operate in high SIL and high demand or continuous mode of operation. For the same system when it shall operate in low demand mode, the requirements are very reasonable, however. The huge discrepancy in feasibility between low demand mode application and high demand mode application for the same system is mathematically clear, however, the practical implications seemed not to be present to the committee members when writing the standard.

A way to mitigate this weakness will be discussed later in this document.

---

1 Constraints: (1) Precisely identifiable unit with a clearly restricted functionality and (2) Demands must cover the full range of normal and abnormal inputs and modes.
3 Approaches to Overcome Important Weaknesses

3.1 Enhance Readability and Usability

Assessors of TÜV Nord and the company exida.com developed independently tools to make the standard(s) and its interpretation more understandable. Both tools focus on different objectives. Whereas the tool from TÜV Nord allocates each requirement to a SIL range, does the exida.com tool SafetyCaseDB merge requirements tracking and IEC 61508 Safety Lifecycle support (Fig. 1) with the safety case approach as described by DStan 00-55.

The exida.com Knowledgebase tool helps development engineers and V&V responsibles to understand and implement each requirement by a typical, generic argument on how to meet the requirement and detailed templates for evidence documents (Fig. 2).

As a characteristic inherited from the Safety Case methodology the tool generates one justification document for many Authorities and opens up the possibility to offload the design teams from compliance work which can be done by safety specialists.
Fig. 2: Relation of Requirements, Arguments and Evidence Documents
The knowledgebase is used in different PES development projects and currently being extended to the requirements of other standards such as EN 954-1, draft IEC 61511-1 and IEC 880 supplement 1. Also outside the safety community, SafetyCaseDB and its underlying V&V Case approach was enhanced and used in the EU Project 11956 „Broadband Access Services Solutions (BASS)“. The EU project with Lucent and the Italian telecom company InfoStrada showed the advantages of a knowledge tool based V&V process also for not-safety-related developments.

3.2 Software Engineering

The following statement based on project experience might seem astonishing considering how long Software Design methods are being described in literature and supported by CASE tools, but “Semi-formal Software Design and well-documented Module Tests are not yet State of the Art in Automation industry”, not in Europe, North America and Japan. UML diagram techniques are used but mostly to enhance textual specifications than in a consistent manner to replace textual specifications. Even if it is accepted that they should be applied to safety projects but development teams do not have experience with these methods from their typical, non-safety projects. The following sections will describe proposals to mitigate the discrepancy to the requirements of IEC 61508-3:

- The careful use of pre-existing software;
- Combine overlapping methods and measures, not described by IEC61508-3
- The use of Software Criticality Analysis to achieve a better SIL allocation to Software.

3.2.1 Safety Dedication Process for Pre-existing Software

Today the responsible product managers typically define the next generation safety systems as a branch of an existing PES product family. This allows and requires the safety development team to re-use the Software platform of the existing products which leads to a controversial situation. The product manager expects a considerable reduction in development time and costs. The safety development team fears the lack of demonstrable quality of the existing software. The emphasis lays on the word “demonstrable”, as one cannot demonstrate that the development of the standard Software platform followed the requirements of IEC 61508-3.

To solve the issue, a Safety Dedication Process for Pre-existing Software is proposed (Fig. 3). The following schematic gives an overview of the relationship between development of the safety system and the safety investigation for the pre-existing software. The process was developed in a nuclear project with industrial partners and TÜV and has shown to be successful.

The safety dedication process is applicable to pre-existing hardware and software products, whereas the schematic emphasizes pre-existing software. The safety dedication process starts in step one with an collection and evaluation of the safety requirements imposed on the pre-existing (software) product by the future application of the safety system and the standards it shall meet. The requirements split into functional safety requirements resulting from how the pre-existing (software) product is used in the new safety system and safety integrity requirements originating from the safety criticality of the pre-existing product’s use in the new safety system.
In step two, the suitability of the pre-existing software product to meet the safety function and safety integrity requirements is evaluated. This is done in two steps: (1) as a paper study comparing the requirements and the pre-existing software product specification; (2) as practical validation tests to demonstrate that mainly the safety function requirements are met by the pre-existing software product. The validation tests should be defined during the paper study.

The safety suitability evaluation should also give some indication on the safety benefits achieved by using the pre-existing software product. The term “Safety Benefit” denotes the benefits an application gains from the use of the pre-existing software product and which would be difficult to achieve if the pre-existing software product would not have been used. Good examples from the context of RT-OS are memory protection with the help of a Hardware Memory Management Unit (MMU) and hence less systematic software errors by better encapsulation by RT-OS processes in particular during software modification. As this term and measure is not introduced in the mentioned standards, no credit is taken in this document.

In step three, a safety assessment of the pre-existing software is executed. The intention of the safety assessment is to evaluate the available safety measures:

- Proven operational experience of the pre-existing software product in other similar, also not safety-related applications;
- Validation efforts demonstrable for the pre-existing software product;
- Test and analysis efforts executed by the user of the pre-existing software product during his safety system V&V which cover the pre-existing software product;

To keep the operating experience of the pre-existing software product out of question, any requests for modification of the pre-existing software product should be avoided.
In **step four**, one specifies and executes or implements additional safety provisions to achieve the required safety integrity:

- Additional validation efforts to be executed by the safety system development team on the pre-existing software product, possibly during their safety system integration tests;
- Safety measures to be implemented by the safety system designer around the pre-existing software product to mitigate safety weaknesses which remained even after the execution of the safety activities listed above. Such safety measures might be implemented by means of a separate safety layer – see below.

The key question for the third step, the safety assessment of the pre-existing software product is “How much is enough for a given criticality of use of the pre-existing software product?”. This might be formally answered by referring to the applicable safety standards (e.g. prEN 50128, IEC 880, IEC 61508), which may, however, turn out to be a knock-out situation for most commercial pre-existing software products (COTS), as their vendors might not be willing to meet the requirements of these standards on verification & validation and proven-in-use demonstration. Hence it is proposed to take into account not only the allocated safety integrity (SIL), but also the criticality of malfunctions of the pre-existing software product in the given application. This can be done by the method of Software Criticality Analysis (SCA), described later in this paper, using worst-case failure modes of the pre-existing software product irrespective of the previous development and V&V efforts. This approach minimizes the need for a detailed investigation of the pre-existing software product itself and is accepted by prEN50128.

This safety dedication process leads to an overlapping of safety measures as summarized by Fig 4.

![Fig. 4. Examples of overlapping Safety Measures](image_url)

In order to facilitate the selection of the appropriate set of safety measures, the definition of groups of failure modes as of table 2 has shown to be helpful:
Table 2. Classification of Failure Modes of Software

<table>
<thead>
<tr>
<th>Failure Mode</th>
<th>ID</th>
<th>Typical mitigation measure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Systematic errors</strong> of the pre-existing software product</td>
<td><strong>SE</strong></td>
<td>Validation of the pre-existing software product with the help of (1) regression test suites or (2) commercial test suites as they are available for RT-OS and Math Libraries defined by standardization committees; AND Proven operating experience;</td>
</tr>
<tr>
<td><strong>Systematic errors</strong> of the pre-existing software product which <strong>appear to be random</strong>, as they reveal themselves only when they are triggered by rare circumstances</td>
<td><strong>RE</strong></td>
<td>Runtime error code checks and assertions by a safety layer implemented by the application specific software;</td>
</tr>
<tr>
<td><strong>Systematic errors</strong> of the pre-existing software product which <strong>appear to be Common Cause system failures</strong>, as they are triggered by external stressors, like frequent events, rare event sequences, memory shortage</td>
<td><strong>CE</strong></td>
<td>Runtime error code checks and assertions at the interfaces of the application specific software; AND Stress testing of the application specific software;</td>
</tr>
<tr>
<td><strong>Errors of the application specific software</strong>, due to misleading specification of the services of the pre-existing software product either being: • incomplete or • prone to misinterpretation or • platform dependent</td>
<td><strong>AE</strong></td>
<td>Integration testing of the application specific software; AND Guidelines for the “safe” use of the pre-existing software product; AND The software criticality analysis of the application specific software shall consider the services of the pre-existing software.</td>
</tr>
</tbody>
</table>

3.2.2 Software Criticality Analysis

Similarly to Hardware where the term Fault Tolerance denotes “the ability of a functional unit to continue to perform a required function in the presence of faults or errors”, the Software Criticality Analysis introduces the term “Fault Tolerance” or “Criticality” in addition to SIL.
C3: **Safety Critical** denotes a function, where a single deviation from the specified function may cause an unsafe situation.

C2: **Safety Relevant** denotes a function, where a single deviation from the specified function cannot cause an unsafe situation, but only in combination with a second independent software error or hardware fault.

C1: **Interference Free** denotes a function, which is not safety critical or safety relevant, but has interfaces with such functions.

C0: **Not-Safety-Related** denotes a function which is not safety critical or safety relevant and has no interaction and no interfaces with such functions.

The Software Criticality Analysis is limited, however, to software units which have clearly restricted functionality(ies) and interfaces. Thus it is most appropriate to software libraries and operating systems incl. drivers and communication stack which have clear interfaces and are used through them.

The benefit of the Software Criticality Analysis is that it allows to reduce the required Software Safety Integrity and thus the necessary safety demonstration effort for lower Software Criticality (C) at the same SIL. This relation is shown in table 3 and justified by the analogy to the Hardware Fault Tolerance in IEC 61508-2, where higher Hardware Fault Tolerance requires less diagnostic effort.

**Table 3. Relation of SIL, Safety Criticality and required Software Safety Integrity**

<table>
<thead>
<tr>
<th></th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIL1</td>
<td>No safety integrity requirements</td>
<td>No safety integrity requirements</td>
<td>SIL1 (recommended)</td>
<td>SIL1</td>
</tr>
<tr>
<td>SIL2</td>
<td>No safety integrity requirements</td>
<td>See remark</td>
<td>SIL1</td>
<td>SIL2</td>
</tr>
<tr>
<td>SIL3</td>
<td>No safety integrity requirements</td>
<td>See remark</td>
<td>SIL2</td>
<td>SIL3</td>
</tr>
<tr>
<td>SIL4</td>
<td>No safety integrity requirements</td>
<td>See remark</td>
<td>SIL3</td>
<td>SIL4</td>
</tr>
</tbody>
</table>

Remark: For Interference free software function three options exist:

1. **No safety integrity requirements**, if the interference freeness can be demonstrated, e.g., by the use of memory protection (MMU);

2. **No safety integrity requirements**, if the implementation languages for the pre-existing software product enforces encapsulation like Modula, ADA, JAVA, C#;

3. **Pointer analysis** for the pre-existing software product for other languages like C and C++.
The Software Criticality Analysis results in much better understanding of any software and good justification for much less investment in activities which are required by the standard but often not done, like semi-formal design methods, use of CASE tools and documented module (coverage) testing. Table 4 shows important design and V&V techniques required for SIL3, C2 software units and the additional effort for SIL3, C3 as of IEC 61508-3.

**Table 4. Difference between Safety Criticality C2 and C3 in SIL3**

<table>
<thead>
<tr>
<th>SIL3</th>
<th>C2</th>
<th>C3 – additional effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="Image" alt="Separation of safety and non-safety software" /></td>
<td><img src="Image" alt="Semi-formal methods using Computer aided tools" /></td>
</tr>
<tr>
<td>Architecture</td>
<td><img src="Image" alt="Structured methods" /></td>
<td><img src="Image" alt="Semi-formal methods" /></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Use of trusted modules" /></td>
<td><img src="Image" alt="Computer aided tools" /></td>
</tr>
<tr>
<td>Detailed design</td>
<td><img src="Image" alt="Semi-formal methods" /></td>
<td><img src="Image" alt="Defensive programming" /></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Design standards" /></td>
<td><img src="Image" alt="Failure detection and diagnosis" /></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Program sequence monitoring" /></td>
<td><img src="Image" alt="Performance testing" /></td>
</tr>
<tr>
<td>Module Testing</td>
<td><img src="Image" alt="Dynamic analysis and testing" /></td>
<td><img src="Image" alt="Stress testing" /></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Functional and black box testing" /></td>
<td>Response timings and memory constraints</td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Boundary value analysis" /></td>
<td>Interface testing</td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Equivalence classes and input partition testing" /></td>
<td></td>
</tr>
<tr>
<td>Integration Testing</td>
<td><img src="Image" alt="Functional and black box testing" /></td>
<td><img src="Image" alt="Performance testing" /></td>
</tr>
<tr>
<td>Validation</td>
<td><img src="Image" alt="Functional and black box testing" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Simulation / modeling" /></td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td><img src="Image" alt="Static analysis" /></td>
<td><img src="Image" alt="Structure based testing (coverage testing)" /></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Control and Data flow analysis" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Design reviews" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Dynamic analysis and testing" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="Image" alt="Test case execution from boundary value analysis" /></td>
<td></td>
</tr>
</tbody>
</table>
The Software Criticality Analysis may be performed using techniques such as Software HAZOP (Hazard and Operability Analysis) [DStan 00-58] which is a systematic design examination to identify what variations from the design intent could occur in the functions, parameters and attributes. The fundamental concept is, that each software function or component has parameters or attributes whose deviations are examined by using guidewords. Table 5 shows typical guidewords. The possible causes and consequences are determined and possible safety measures specified.

<table>
<thead>
<tr>
<th>Guideword</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No part of the design intention is achieved</td>
</tr>
<tr>
<td>More</td>
<td>A quantitative increase</td>
</tr>
<tr>
<td>Less</td>
<td>A quantitative decrease</td>
</tr>
<tr>
<td>As well as</td>
<td>All design intent achieved but with additional results</td>
</tr>
<tr>
<td>Part of</td>
<td>Only some of the intention is achieved</td>
</tr>
<tr>
<td>Other than</td>
<td>A result other than the original intention is achieved</td>
</tr>
<tr>
<td>Early</td>
<td>Something happens earlier than expected</td>
</tr>
<tr>
<td>Before</td>
<td>Something happens before the expected step</td>
</tr>
<tr>
<td>Never</td>
<td>Something happens never</td>
</tr>
<tr>
<td>Late</td>
<td>Something happens later than expected</td>
</tr>
<tr>
<td>After</td>
<td>Something happens after the expected step</td>
</tr>
</tbody>
</table>

The Software HAZOP does not consider the likelihood of failures of the pre-existing software product to happen. This leads to unnecessarily controversial discussions with the development engineers. Hence we recommend to determine a likelihood of the failure mode to reside still undetected in the pre-existing software product (Table 6). The likelihood is not based on quantitative reasoning but reflects a qualitative engineering judgment.

4 Conclusion

IEC 61508 is here and it is a big success. Buyers of Programmable Electronic Systems and Authorities see it as a major reference to reduce their uncertainty on complex systems in their safety applications. The learning curve is steep and requires quite some investment upfront in learning and coaching to select and set up the appropriate safety techniques. Software tools are available which help to understand and answer the requirements and support the design and V&V processes. As with any new specification, the standard leaves room for improvement. Methods and techniques are available for software safety design and verification which are not specified by the standard but have great benefit in addressing its objectives and requirements while meeting the today needs for software re-use in order to cut the time to market.
### Table 6. Proposal for Qualitative Definition of Software Error Likelihood

<table>
<thead>
<tr>
<th>Likelihood of Failure</th>
<th>Qualitative definition</th>
<th>Consequence for the Software HAZOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>The pre-existing software product implementation or its failure mode are platform dependent. Thus the pre-existing software product could not sufficiently be tested or analyzed to uncover this particular failure mode. The depth of design verification or test coverage of the pre-existing software product cannot be demonstrated and Proven-in-Use as a measure on its own is deemed to be insufficient for the required SIL.</td>
<td>The potential effect of the failure mode on the application specific software shall always be analyzed.</td>
</tr>
<tr>
<td>Moderate</td>
<td>The depth of design verification or test coverage of the pre-existing software product cannot be demonstrated, but Proven-in-Use is deemed to be adequate for the required SIL and software criticality.</td>
<td>The potential effect of the failure mode on the application specific software should be analyzed. The potential effect of the failure mode on the application specific software shall always be analyzed, if the particular service / function is used in a safety-critical function (C3) of the application specific software.</td>
</tr>
<tr>
<td>Low</td>
<td>The pre-existing software product should have already been well tested to uncover this potential failure mode. If doubts exist, the use of the pre-existing software product may be inadequate at all.</td>
<td>Analysis of the potential effect of the failure mode on the application specific software is not required.</td>
</tr>
</tbody>
</table>
References

[DStan 00-55] Requirements for Safety Related Software in Defence Equipment
[DStan 00-58] HAZOP Studies on Systems Containing Programmable Electronics
[Faller 2001] Hardware Architectures and Common Cause; June 2001
(can be requested from the author: Rainer.Faller@exida.com)
[Wiegerinck 1999] Interkama / ISATech 1999; Compliance to IEC 61508 (/ IEC 61511) Classification and Implementation of Instrumented Protective Functions – The SHELL approach; Jan Wiegerinck; SHELL Global Solutions
About the Design of Distributed Control Systems: The Quasi-Synchronous Approach

Paul Caspi¹, Christine Mazuet², and Natacha Reynaud Paligot²

¹VERIMAG♣ 2 rue de Vignate, F-38610 Gières
   caspi@imag.fr
²Schneider Electric, Usine M3, F-38050 Grenoble Cedex 9
   {christine_mazuet, natacha_reynaud-paligot}@mail.schneider.fr

Abstract. The European project Crisys aims at improving and formalizing the actual methods, techniques and tools used in the industries concerned with process control, in order to support a global system approach when developing Distributed Control System. This paper focuses on the main result of the Crisys project: the quasi-synchronous approach which is based on the synchronous language Lustre-Scade. The quasi-synchronous methodology provides (1) a complete framework consistent with usual engineering practices for both programming, simulating, testing a distributed system and (2) a robustness properties checker so as to ensure the behavior preservation during the distributed implementation. Both elements are based on a solid theoretical basis.

1 Introduction

Developing Distributed Control System is a major industrial concern since those systems are more and more complex and involved in many safety critical application field. The distribution feature of these systems is not without consequences on both the development process and the exploitation of the system: the global behavior of the system is more complex since distribution introduces new operating modes — abnormal modes, when a computing site is down for instance— and questions about the synchronization of the different computing sites. Distributed Control Systems (DCS) are hard to design, debug, test and formally verify. These difficulties are closely related to a lack of global vision at design time. Moreover, the implementation would be eased using automatic methods of distribution which guarantee that the behavior of the whole system is preserved.

To face up to these difficulties engineers have developed solutions of their own. Their solutions are essentially pragmatic and based on engineering rules. But a theoretical basis is lacking if we want formally to understand, design and verify Distributed Control Systems when applied to critical fields.

The European project Crisys originates from this industrial need. The overall goal of the Crisys project is to improve, unify and formalize the actual methods, techniques

♣ VERIMAG is a joint laboratory of Université Joseph Fourier (Grenoble 1), CNRS and INPG.
 http://borneo.gmd.de/~ap/crisys/

This work has been partially supported by Esprit Project CRISYS (EP 25514).
and tools used in the industries concerned with the process control, in order to support a global system approach when developing Distributed Control System. This paper focuses on the main result of the Crisys project, the quasi-synchronous approach which is based on the synchronous language Lustre [1] and the associated tool Scade [2]. This approach is dedicated to a special class of DCS: in the Control field, most of DCS are organized as several periodic processes, with nearly the same period, but without common clock, and which communicate by means of shared memory through serial links or field busses. This class of DCS is quite clearly an important one in the field and thus deserves special attention.

The paper is organized as follows: section 2 presents an overview of the Crisys methodology based on the quasi-synchronous approach. Then, section 3 briefly describe the Lustre-Scade tool-set for designing distributed systems. In section 4, we focus on the robustness properties which guarantee that the centralized behavior of the system is preserved when distributing the system according to the chosen architecture. Section 4 describes the application of the Crisys methodology to an industrial case study. Finally, section 6 concludes with future work.

2 Overview

2.1 Industrial Practices

The Lustre-Scade language is largely and successfully applied to the development of distributed control systems [3] [4] [5]. But so far, the engineers make use of Lustre-Scade to design single components of a DCS. Schematically, the industrial software development proceeds as follows (Fig. 1):

- The specification phase involves both the functional description —i.e. the behavior of the whole system independently of its architecture— and the distribution protocol which specifies the physical implantation of the functional components. So far, the solution to design robust distributed systems —i.e. whose functional behavior is preserved when distributing it— are pragmatic and based on the engineers know-how.
- Each component is developed separately with Lustre-Scade. The global view of the system is no longer preserved. Moreover, there is usually a breaking in the tool chain between this step and the previous one.
- Finally, pieces of code resulting from the previous step are plug into the physical target and connected by means of network (e.g. [6]).

The goal of the Crisys project is to improve this development process based on Lustre-Scade, by formalizing the industrial practices and providing support of tools.

2.2 The CRISYS Methodology

The methodology defined within the Crisys project is shown on Fig. 2.

1. From the functional specification, a Lustre-Scade model of the global system is developed. At this stage, this functional model can be simulated, formally verified and tested.
2. The second step consists in completing the functional model with the distribution protocol. Then, the resulting architecture is checked by means of the robustness properties analyzer. This tool aims at guaranteeing that the behavior of the distributed system is consistent with the behavior of the centralized one. The analysis is based on three robustness properties: stability, order-insensitivity, and confluence (§4).

3. A distribution scheme being acceptable, it is possible to test, simulate, formally verify the distributed system in a realistic way by means of the environment emulation library. It is important to note that the same tools applied to the centralized model and to the distributed one allowing the comparison of their behavior.

4. Finally, the code corresponding to each component is generated together with some communication elements provided by the communication library for target.

An application of this methodology is presented in section 5.

The valuable consequences of the Crisys methodology are:

-The global view of the distributed system is preserved as long as possible during design. It can be simulated and tested as a whole.

-The robustness properties analyzer based on theoretical foundations formalizes the pragmatic and intuitive solutions achieved by engineers to design robust DCS.
The same framework can be used for programming, simulating, testing and proving properties of a distributed system. This result makes the comparison between the behavior of the centralized and the distributed system possible. The Crisys work has first focused on the use of Lustre-Scade for designing a DCS as a whole, i.e. for describing the Scade distributed model (§3.3). Then, the second step has concentrated on the robustness properties analysis (§4).

3 Background

3.1 The Lustre Language and the Scade Tool

Lustre [1] is a synchronous data-flow language. Each expression or variable denotes a flow, i.e., a function of discrete time. The Lustre equation $x=2*y+z$ means: “at each instant $t$, $x(t)=2*y(t)+z(t)$”. Lustre provides a special operator “previous” to express delays: “$y=pre(x)$” means that at each time $t\neq 0$ we have $y(t)=x(t-1)$, while the value of $y$ at time 0 is undefined. To initialize variables, Lustre provides the “followed by” operator: “$z=x\rightarrow y$” means that $z(0)=x(0)$ and $z(t)=y(t)$ for each time $t\neq 0$.

A Lustre program is structured into nodes. A node contains a set of equations and can be elsewhere used in expressions. It may be that slow and fast processes coexist in a given application. A sampling (or filtering operator) when allows fast processes to communicate with slower ones. Conversely, a holding mechanism, current allows slow processes to communicate with faster ones.

Scade\(^1\) (formerly SAGA [2]) is a software development environment based on Lustre, which provides a graphic editor. Its main features are the top-down design method, the data-flow network interpretation, and the notion of activation condition.

An example of Scade diagram is given on Fig. 3. CONTROL is a cyclic program which reads sensors and controls actuators. Its inputs and outputs are sampled according to the boolean condition clock: intuitively, if clock is true then CONTROL computes its outputs, else the outputs are maintained to their previous values. Default values are required in case clock is false at the very first cycle.

![Fig. 3. Example of Scade diagram](image)

The Scade environment includes an automatic C code generator and a simulator. It is also connected to several tools (§3.2).

---

\(^1\) Scade is commercialised by the Telelogic company.
3.2 The Lustre-Scade Tool-Set

Several tools have been developed to improve and facilitate the design and the verification of Lustre-Scade programs. For example, Lesar [7] and Lucifier2 [8] for formal verification, Matou for describing Mode-Automata [9], Lurette [10] for automatic test cases generation. Scade can be also connected to ISG2 [11] for performance validation.

Let us concentrate on the automatic generation of test sequences with Lurette. The automatic generation of test cases follows a black box approach, since the program is not supposed to be fully known. It focuses on two points [10]: (1) generating relevant inputs, with respect to some knowledge about the environment in which the system is intended to run; (2) checking the correctness of the test results, according to the expected behavior of the system. The Lustre synchronous observers [3] describing assumptions on the environment are used to produce realistic inputs; synchronous observers describing the properties that the system should satisfy (§2.3.1) are used as an oracle, i.e. to check the correctness of the test results. Then, the method consists in randomly generating inputs satisfying the assumptions on the environment [10].

The Lurette tool takes as input both observers —one describing the assumptions and one describing the properties— written in Lustre-Scade, and two parameters: the number of test sequences and the maximum length of the sequences. An experimentation of Lurette is presented in section 5.

3.3 The Quasi-Synchronous Approach

The above language and tools accurately match the needs of single cyclic components. But how can they be used to design a distributed system as a whole? The first step of the Crisys work aimed at formalizing the description of a DCS by means of the Lustre-Scade language [13].

First let us remind ourselves the main features of the quasi-synchronous class of DCS: process behave periodically, they all have nearly the same period but no common clock and they communicate by means of shared memory. These features can be formalized by means of the Lustre-Scade primitives (Fig. 4):

Each processes has got its own clock represented by an activation condition. For example, on Figure3, process SI is activated each time its clock cl is true.

Shared memories are modelled through both the activation condition and delays (pre, ->).

Finally hypothesis on clocks can be implemented through a Lustre-Scade program: the quasi-synchronous program generates clocks with nearly the same period (§5.3.3, Fig. 12) This is one of the component of the environment emulation library (Fig. 2).

---

2 Partly developed within the framework of the Crisys project.
3 Synchronous observers are acceptors of sequences [12].
4 Towards a Robust Distribution

Given the Lustre-Scade model of the distributed system, which additional checks have to be performed so as to ensure the behavior preservation during the implementation? Three constraints —called robustness properties— have been identified in order to guarantee that the behavior of the distributed system is the same as the centralized one. These checks are implemented through a tool —the robustness properties analyzer— which is one of the key element of the Crisys methodology. In this chapter, we present in an informal way the three robustness properties. The theoretical details can be found in [14] [15] [16].

4.1 Stability

It is likely indeed that distributed programs will have to run faster in order to produce behaviors comparable to those of centralized programs. But running a synchronous program faster on the same inputs will in general deeply modify its behavior. This is why we may expect it easier to distribute stable systems rather than unstable ones, stable systems being those that can run faster without too much changing their behaviors.

In other words, a stable system will stabilize when the inputs do not change. Figure 5 gives an example of non-stable system: when \( u \) remains true, the output \( x \) is indefinitely oscillating between true and false. Let us now suppose a redundant system involving two sub-systems defined by the equation of Figure 3. The oscillation made the comparison of both results meaningless.

\[
x = u \text{ and } (\text{false } \rightarrow \text{not pre } x) ;
\]

Fig. 5. Example of a non stable system
4.2 Order Insensitivity

Another feature of distributed systems is that their components are not computed in a parallel synchronous fashion but in a sequential (chaotic ordered) way. A system is order-insensitive if its behavior does not depend on the order computations are performed. Figure 6 gives an example of an order sensitive system. As regards the centralized behavior (Fig 6.a), the output y reaches the true value because the input u is true and the previous value of x is false. Let us now assume that computations of x and y are performed on two different processors running with different time cycles. If the x value is computed and sent to the other processor before y is computed (Fig. 6.b), then y can no more reach the true value because its calculation refers to the latest value of x which is now true.

\[
\begin{align*}
x &= u \text{ or } (\text{false } \rightarrow \text{pre } x) \\
y &= u \text{ and } (\text{false } \rightarrow \text{pre not } x) \text{ or } (\text{false } \rightarrow \text{pre } y)
\end{align*}
\]

(a)Centralized behavior  (b)Distributed behavior

Fig. 6 . Example of an order sensitive system

A stronger property called state decoupling [14] is satisfied when each component depends only on its internal state.

4.3 Confluence

Another desirable property for distribution is confluence. It means that input changes can be arbitrarily composed while yielding the same final state. The order the inputs are read does not have to imply different behaviors. An example of a non confluent system is given on Figure 7. The outputs x and y are obviously equal (when they are computed in a centralized manner). But if the inputs u and v are sampled according to the dotted line then x and y differ from each other. The centralized behavior is no longer preserved.

\[
\begin{align*}
x &= u \text{ and not } v \text{ or } (\text{false } \rightarrow \text{pre } x) \\
y &= u \text{ and not } v \text{ or } (\text{false } \rightarrow \text{pre } y)
\end{align*}
\]

Fig. 7 . Example of a non confluent system
However, confluence is a very restricted property and we cannot limit ourselves to distributing confluent functions. We may need to strengthen this definition by considering \textit{local confluence} [14].

5 Case Study

The Crisys methodology (§2.2) is now illustrated on a real case study from Schneider. Through this experimentation, our aim is to check the feasibility and the benefits of the quasi-synchronous approach based on Lustre-Scade.

5.1 Introduction

The Water Level Control System (WLCS) is a system controlling the water level in a steam generator. This system is aimed to be implemented in power plants (nuclear or thermal). Basically, the WLCS operates on two valves so that the water level is unchanged. Several sensors are present all along the steam generator to measure the water level, the flow, the temperature and the thermal power.

The WLCS is a typical loop system. It is made of three steps (Fig. 8):
- the water level control loop that provides a water flow set point,
- the water flow control loop that provides the valves position set point,
- the valves position control loop that controls the valves.

One of the main requirement is that the valves have to be controlled in a smoothly way in order to avoid discrepancies. During the experimentation, a particular attention has been taken on the switching between the automatic and the manual mode, since this change may imply discrepancies.
The WLCS has been developed with SCADE using the CRISYS methodology (Fig. 2). The experimentation has been conducted through different steps:

At first, the WLCS has been designed and simulated in the centralized way.

Then a distributed architecture has been proposed and analyzed.

Finally, the distributed system has been simulated and its behavior has been compared to the centralized one.

### 5.2 The Centralized System: Design and Simulation

The centralized system has been designed with the SCADE tool according to the functional view showed on Figure 8. Moreover, in order to simulate the system as if it was physically implemented, the behavior of the different sensors has been designed, i.e. the system is simulated in closed loop (Fig. 9).

The automatic generation of test sequences, Lurette, has been used to simulate the system.

![Fig. 9. The closed loop system](image1)

![Fig. 10. Example of result](image2)

An example of results is given on Figure 10. We can see that after the initialization phase, the valve opening stabilizes at 25% after 2000 cycles (i.e. 500 seconds).

### 5.3 The Distributed System

**Architecture analysis.** The architecture of the system has been defined by the client for performance reasons. The system is made of two sub-systems which communicate with each other (Fig. 8):

- the first sub-system involves the water level control loop and the water flow control loop,
- the second sub-system involves the valves position control loop.

In order to guarantee that the behavior of the distributed system is preserved, this architecture has been analyzed with the robustness properties analyzer (see §4.2). The result of the tool is that the WLCS is stable, order-insensitive and confluent as far as the proposed architecture is concerned.
Scade design. According to the quasi-synchronous approach (§3.3), each sub-system has got its own clock representing its own cycle. The WLCS is composed of two sub-systems which have different clocks (Fig. 11):

- the water level control loop and the water flow control loop have the same clock (CLK1),
- the water position control loop has a different clock (CLK2).

**Fig. 11. SCADE distributed model**

Simulation. The goal of the simulation is to check the behavior of the distributed system in a realistic way. Clocks are generated according to the quasi-synchronous hypothesis (i.e. periodic real time clocks of each process are subject to drifts) by means of the environment emulation library. An example of the clocks used for the two WLCS’s sub-systems is given on Figure 12. These clocks are pessimistic since data can be lost.

**Fig. 12. Quasi-synchronous clocks**
5.4 Results and Comparisons

The behaviors of the centralized system (Fig. 13) and the distributed system (Fig. 14) are similar: first, the low flow valve opens up to 30% and then stabilizes around 25%. In manual mode, the operator increases the opening set point. When coming back to the automatic mode, the valve opening oscillates and then stabilizes again at 25%. The control is performed in a smoothly way as required for the centralized and the distributed systems.

As regards the distributed case, we can note that the time response is slower due to the communication delays between the two sub-systems during the simulation.

6 Conclusion and Future Work

The experimentation shows the feasibility and the benefit of the quasi-synchronous methodology. An additional experimentation on a case study from the aircraft industry enforces this conclusion. The quasi-synchronous methodology provides:

- a global view of the Distributed Control System which can be designed and simulated within the same environment, in consistency with the usual engineering practices;
- an automatic robustness analyser which aims at guaranteeing that the behaviour will be preserved when distributing the system according to the target architecture.

These two points are key elements to reduce the industrial development costs.

The next steps of the work are twofold:
- some tools need to be improved so that they can easily be integrated in the industrial development process;
- the experimentation on the Schneider case study will be continued until the final implementation of the generated code.
Acknowledgement

The authors would like to thank all the partners involved in the Crisys project.

References

Dependability Evaluation
From Functional to Structural Modelling

Cláudia Betous-Almeida and Karama Kanoun

LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 - France
{almeida,kanoun}@laas.fr

Abstract. The work presented in this paper is devoted to the definition of a dependability modelling approach for the selection process of instrumentation and control systems (I&C) in power plants. We show how starting from functional specifications, a functional-level model can be transformed into a dependability model taking into account the system's architecture, following a progressive and hierarchical approach. This approach is illustrated on simple examples related to a specific architecture of an I&C system.

1 Introduction

Dependability evaluation plays an important role in critical systems’ definition, design and development. Modelling can start as early as system functional specifications, from which a high-level model can be derived to help analysing dependencies between the various functions. However, the information that can be obtained from dependability modelling and evaluation becomes more accurate as more knowledge about system implementation is integrated into the models. The aim of this paper is to show how starting from functional specifications, a functional-level model can be transformed into a dependability model taking into account the system’s architecture, using a progressive modelling approach. The modelling approach has been applied to three different instrumentation and control systems (I&C) in power plants, to help selecting the most appropriate one. Due to space limitations, in this paper we illustrate it on a small part of one of them.

The remainder of the paper is organised as follows. Section 2 gives the context of our work. Section 3 is devoted to the presentation of the modelling approach. Section 4 presents a small example of application of the proposed approach to an I&C system and Section 5 concludes the paper.

2 Context of Our Work

The process of defining and implementing an I&C system can be viewed as a multi-phase process (as illustrated in Figure 1) starting from the issue of a Call

© Springer-Verlag Berlin Heidelberg 2001
for Tenders by the stakeholder. The call for tenders gives the functional and non-functional (i.e., dependability) requirements of the system and asks for candidate contractors to make offers proposing possible systems/architectures satisfying the specified requirements. A preliminary analysis of the numerous responses by the stakeholder, according to specific criteria, allows the pre-selection of two or three candidate systems. At this stage, the candidate systems are defined at a high level. They are usually based on Commercial-off-the-Shelf (COTS) components and the application software is not entirely written. The comparative analysis of the pre-selected candidate systems, in a second step, allows the selection of the most appropriate one. Finally, the retained system is refined and thoroughly analysed to go through the qualification process. Dependability modelling and evaluation constitute a good support for both the selection and the refinement processes, thorough analysis and preparation of the final system’s qualification. The main purpose of our work is to help the stakeholder in this modelling process. To this end, we have defined a rigorous, systematic and hierarchical modelling approach that can be easily used to select an appropriate architecture and to model it thoroughly. Thus this approach can be used by any system’s developer.

![Fig. 1. Various steps of I&C definition process](image)

3 Modelling Approach

Our modelling approach follows the same steps as the development process: It is also performed in three steps as described in Figures 1 and 2:

Step A. Construction of a functional-level model based on the system’s specifications;

Step B. Transformation of the functional-level model into a high-level dependability model, based on the system’s architecture. There is one for each pre-selected candidate system;

Step C. Refinement of the dependability model, based on the detailed architecture of the retained system.

Modelling is based on Generalised Stochastic Petri Nets (GSPN) due to their ability to cope with modularity and model refinement [1]. The GSPN model is processed to obtain the corresponding Markov chain. Dependability measures (i.e., availability, reliability, safety, ...) are obtained through the processing of the Markov chain, using an evaluation tool such as SURF-2 [3].
3.1 Functional-Level Model

The system’s functional-level model is the starting point of our method. This model is independent from the underlying system’s architecture. Hence it can be done even before the call for tenders, by the stakeholder.

The system’s functional-level model is formed by places which represent the possible states of functions. For each function, the minimal number of places is two (Fig. 3): One which represents the function’s nominal state (F) and the other its failure state ($\overline{F}$). Between these two states, we have the events that manage changes from F to $\overline{F}$ and vice-versa. These events are inherent to the system’s structure that is not specified in this step as it is not known yet. We call the model that contains these events and the corresponding places, the link model ($M_L$). Note that the set {$F, M_L, \overline{F}$} that constitutes the system’s GSPN model, will be completed once the architecture system is known\(^1\).

![Functional-level model related to a single function](image)

Most of the times though, systems perform more than one function. In this case we have to look for dependancies between these functions due to the communication between them. We distinguish two degrees of dependancy. Figure 4 illustrates the two types of functional dependancy between two functions $F_1$ and $F_2$. $F_3$ is independent of both $F_1$ and $F_2$.

Case (a) Total dependancy – $F_2$ depends totally on $F_1$, noted $F_2 \leftrightarrow^* F_1$. In this case, if $F_1$ fails, $F_2$ also fails, i.e. ($M(F_1) = 1$) $\Rightarrow$ ($M(F_2) = 1$), where $M(F)$ represents the marking of place $F$;

\(^1\) This modelling approach is applicable in the same manner when there are several failure modes per function.
3.2 Link Model

The link model gathers the set of states and events related to the architectural behaviour of the system. The first step in constructing this model consists on the identification of the components associated with the system’s functions. For modelling purposes, consider the following complete set of cases:

**Case A.** One function: In this case, several situations may be taken into account. A function can be done by:
- A.1. A single software component on a single hardware component;
- A.2. Several software components on a single hardware component;
- A.3. A single software component on several hardware components;
- A.4. Several software components on several hardware components;

**Case B.** Several functions: Again two situations can take place:
- B.1. The functions have no common components;
- B.2. The functions have some common components.

To illustrate the given situations, we will consider a simple example for each case. Here we give only an overview of the structure of the link model. Note that the structural models presented in this section are not complete. More information is given in sections 3.3 et 3.4.

**Case A.** Case of a single function.

**A.1.** Let us suppose function F carried out by a software component S and a hardware component H – Figure 4. Then, F and F markings depend upon the markings of the hardware and software component models. More specifically:
- F’s up state is the combined result of H’s up state and S’s up state.
- F’s failure state is the result of H’s failure or S’s failure.

The behaviour of H and S is modelled by the so-called structural model (MS) and then it is connected to F through an interface model referred to as MI. The link model (ML) is thus made up of the structural model (MS) and of the interface model (MI): ML = MS + MI. This interface model connects hardware and software components with their functions by a set of immediate transitions. Note that there is only one interface model but to make its representation easier, we split it into two parts: An upstream part and a downstream part.

**Case A.2.** Consider function F done by two software components S₁ and S₂ on a hardware component H, in which case we have to consider two situations:

- S₁ and S₂ redundant (Fig. 6(a))
  i. F’s up state is the combined result of H’s up state and S₁ or S₂’s up states:

  \[ \mathcal{M}(F) = 1 \equiv (\mathcal{M}(H_{\text{ok}}) = 1 \land [\mathcal{M}(S_{1\text{ok}}) = 1 \lor \mathcal{M}(S_{2\text{ok}}) = 1]) \]

  ii. F’s failure state is the result of H’s failure or S₁’s failure and S₂’s failure:

  \[ \mathcal{M}(\overline{F}) = 1 \equiv (\mathcal{M}(H_{\text{def}}) = 1 \lor [\mathcal{M}(S_{1\text{def}}) = 1 \land \mathcal{M}(S_{2\text{def}}) = 1]) \]

- S₁ in series with S₂ (Fig. 6(b))
  i. F’s up state is the combined result of H, S₁ and S₂’s up states:

  \[ \mathcal{M}(F) = 1 \equiv (\mathcal{M}(H_{\text{ok}}) = 1 \land [\mathcal{M}(S_{1\text{ok}}) = 1 \land \mathcal{M}(S_{2\text{ok}}) = 1]) \]
ii. F’s failure state is the result of H’s failure or $S_1$ or $S_2$’s failure:
\[
\mathcal{M}(F) = 1 \equiv (\mathcal{M}(H_{def}) = 1 \lor \mathcal{M}(S_{1def}) = 1 \lor \mathcal{M}(S_{2def}) = 1)
\]

A.3. The case of function F done by a single software on several hardware components, is essentially similar to the previous case;

A.4. Suppose function F done by a set of N components:
   i. If all components, under the same conditions, have different behaviours, then the structural model will have N initial places. This case corresponds to a generalisation of Case A.1.
   ii. If some of the N components, under the same conditions, have exactly the same behaviour, their structural models are grouped. In this case, the structural model will have Q initial places ($Q < N$).

Case B. Consider two functions (the generalisation is straightforward) and let \( \{C_{1i}\} \) (resp. \( \{C_{2j}\} \)) be the set of components associated to $F_1$ (resp. $F_2$).

B.1. $F_1$ and $F_2$ have no common components, \( \{C_{1i}\} \cap \{C_{2j}\} = \emptyset \). The interface models related to $F_1$ and $F_2$ are built separately in the same way as explained for a single function.

B.2. $F_1$ and $F_2$ have some common components, \( \{C_{1i}\} \cap \{C_{2j}\} \neq \emptyset \). This case is illustrated on a simple example:
   - $F_1$ done by three components: A hardware component $H$ and two software components $S_{11}$ and $S_{12}$. $F_1$ corresponds to case (a) of Figure 6.
   - $F_2$ done by two components: The same hardware component $H$ as for $F_1$ and a software component $S_{21}$. $F_2$ corresponds to Case A.1. of Figure 3.

Their model is given in Figure 4. It can be seen that i) both interface models ($M_{II}$ and $M_{I2}$) are built separately in the same way as done before, and ii) in the global model, the common hardware component $H$ is represented only once by a common component model.
3.3 Interface Model

The interface model $M_I$ connects the system components with their functions by a set of transitions. This model is a key element in our approach. It has been defined to be constructed in a systematic way in order to make the approach re-usable and to facilitate the construction of several models related to various architectures. Moreover, it has been defined in formal terms. The main rules are stated in an informal manner in this paper.

Both parts of the $M_I$ have the same number of immediate transitions and the arcs that are connected to these transitions are built in a systematical way:

- **Upstream $M_I$**: It contains one function transition $t_F$ for each series (set of) component(s) to mark the function’s up state place and one component transition $t_{C_x}$ for each series, distinct component that has a direct impact on the functional model, to unmark the function’s up state place.
  - Each $t_F$ is linked by an inhibitor arc to the function’s up state place, by an arc to the function’s up state place and by one bidirectional arc to each initial (ok) component place;
  - Each $t_{C_x}$ is linked by an arc to the function’s up state place and by one bidirectional arc to each component failure place.

- **Downstream $M_I$**: It contains one function transition $t'_F$ for each series (set of) component(s) to unmark the function’s failure state place and one component transition $t'_{C_x}$ for each series, distinct component that has a direct impact on the functional model, to mark the function’s failure state place.
  - Each $t'_F$ is linked by an arc to the function’s failure state place and by one bidirectional arc to each initial (ok) component place;
  - Each $t'_{C_x}$ is linked by an inhibitor arc to the function’s failure state place, by an arc to the function’s failure state place and by one bidirectional arc to each component failure place.
3.4 Structural Model

In order to build the interface between the functional and the structural models, we need to identify the components implementing each function, and thus the initial places as well as their failure state places of the structural model.

The structural model can be built by applying one of the many existing modular modelling approaches (see e.g., [4, 5, 6, 7]).

To complete the above examples, let us consider the simple case of Figure 5. The associated structural model is given in Figure 8 in which the $S_{def}$ place of Figure 5 corresponds to either place $S_{ed}$ or $S_{ri}$. The following assumptions and notations are used:

- The activation rate of a hardware fault is $\lambda_h$ (Tr1) and of a software fault is $\lambda_s$ (Tr6);
- The probability that a hardware fault is temporary is $t$ (tr1). A temporary fault will disappear with rate $\varepsilon$ (Tr2);
- A permanent hardware fault (resp. software) is detected by the fault-tolerance mechanisms with probability $d_h$ (resp.$d_s$ for software faults). The detection rate is $\delta_h$ (Tr3) for the hardware and $\delta_s$ (Tr7) for the software;
- The effects of a non detected error are perceived with rate $\pi_h$ (Tr4) for the hardware and rate $\pi_s$ (Tr8) for the software;
- Errors detected in the hardware component require its repair: repair rate is $\mu$ (Tr5);
- Permanent errors in the software may necessitate only a reset. The reset rate is $\rho$ (Tr9) and the probability that an error induced by the activation of a permanent software fault disappears with a reset is $r$ (tr7);
- If the error does not disappear with the software reset, a re-installation of the software is done. The software’s re-installation rate is $\sigma$ (Tr10).

Note that a temporary fault in the hardware may propagate to the software (tr11) with probability $p$. We stress that when the software component is in place $S_{ed}$ or $S_{ri}$, it is in fact not available, i.e., in a failure state.

Also when the hardware is in the repair state, the software is on hold. The software will be reset or re-installed as soon as the hardware repair is finished. Due to the size of the subsequent model, this case is not represented here.

4 Application to I&C Systems

An I&C system performs five main functions: Human-machine interface (HMI), processing (PR), archiving (AR), management of configuration data (MD), and interface with other parts of the I&C system (IP). The functions are linked by the partial dependencies given in column 1 of Table 1.

Taking into account the fact that a system’s failure is defined by:

$$\mathcal{M}(\text{HMI}) = 1 \lor \mathcal{M}(\text{PR}) = 1 \lor \mathcal{M}(\text{IP}) = 1$$

the above dependencies can be simplified as given in column 2 of Table 1.
Table 1. Functional dependancies of I&C systems

<table>
<thead>
<tr>
<th>Function dependancies</th>
<th>Simplified funct. dependancies</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMI ← {PR, AR, MD}</td>
<td>HMI ← {AR, MD}</td>
</tr>
<tr>
<td>PR ← {HMI, MD, IP}</td>
<td>PR ← MD</td>
</tr>
<tr>
<td>AR ← {HMI, MD}</td>
<td>AR ← MD</td>
</tr>
<tr>
<td>IP ← {PR, MD}</td>
<td>IP ← MD</td>
</tr>
</tbody>
</table>

These relations are translated by the functional model depicted in Figure 9.

To illustrate the second step of our modelling approach, we consider the example of the I&C system used in [2]. This system is composed of five nodes connected by a Local Area Network (LAN). The mapping between the various nodes and functions is given in Figure 10. Note that while HMI is executed on four nodes, node 5 runs three functions. Nodes 1 to 4 are composed of one computer each. Node 5 is fault-tolerant: It is composed of two redundant computers. The structural model of this I&C is built as follows:

- Node 1 to Node 3 – in each node, a single function is achieved by one software component on a hardware component. Its model is similar to the one presented in Figures 5 and 8.

Fig. 8. Structural model of a software and a hardware components
• Node 4 – has two independent functions. Its structural model will be similar to the one depicted in Figure 7, followed by a model slightly more complex than the one of Figure 8.

• Node 5 – is composed of two hardware components with three independent functions each. Its structural model is more complex than the previous one due to the redundancy. A part of this model has been presented in 2.

• LAN – the LAN is modelled at the structural level by the new structural dependencies that it creates.

5 Conclusions

In this paper a three step modelling approach has been presented. This approach is progressive and hierarchical and can easily be used to select and thoroughly
model an appropriate architecture. The functional-level and the structural models are linked by an interface model that is constructed in a formal way. This interface model plays a central role in our modelling approach.

Although we have presented in this paper the application of our approach to a small part of an I&C system, the approach has been applied to two other I&C systems to identify their strong and weak points.

The work is still in progress. In particular, the refinement of the dependability model with the formal definition of refinement rules is under study. This will help in the third step of the modelling approach for thorough analysis of the retained system.

References

Tuning of Database Audits to Improve Scheduled Maintenance in Communication Systems

Stefano Porcarelli¹, Felicita Di Giandomenico², Amine Chohra²,* and Andrea Bondavalli³

¹ Univ. of Pisa, Computer Engineering Dep., Via Diotisalvi 2,I-56126, Pisa, Italy
stefano.porcarelli@guest.cnuce.cnr.it

² IEI/CNR, Via Moruzzi 1, I-56100, Pisa, Italy
{digiandomenico, chohra}@iei.pi.cnr.it

³ Univ. of Firenze, Dip. Sistemi e Informatica, V. Lombroso 6/17, I-50134, Firenze, Italy
andrea.bondavalli@cnuce.cnr.it

Abstract. To ensure the consistency of database subsystems involved in communication systems (e.g., telephone systems), appropriate scheduled maintenance policies are necessary. Audit operations, consisting in periodic checks and recovery actions, are typically employed in databases to cope with run time faults which may affect the dependability and quality of service of the overall system. This paper aims at investigating on appropriate tuning of audit operations, so as to find optimal balances between contrasting requirements, namely satisfactory database availability and low overhead due to audits. For this purpose, a methodology to analyse the behaviour of the database under scheduled maintenance is here suggested. Analytical models, essentially based on Deterministic and Stochastic Petri Nets (DSPN), are defined and analysed, in terms of dependability indicators. A sensitivity analysis wrt to the most affecting internal and external parameters is also performed on a case study.

1 Introduction

The problem of protecting data used by applications during their execution, against run-time corruption, has long been recognised to be a critical aspect highly impacting on the reliability/availability of systems relying on such internal database. Communication systems, such as telephone systems, are today-typical systems suffering from this problem, especially when a wireless environment is involved, which makes the data more prone to corruption. Indeed, these systems need to keep trace of resource usage status and of users data for correctly setting up and managing user calls. For this purpose, a database is included, where data are organised in such a way to capture the relationships existing among them. Data corruption may result in the delivery of a wrong service or in the unavailability of the service, with (possibly heavy) consequences on the quality of service perceived by users. Effective mechanisms to detect and recover from data corruption are then necessary; typically, audit operations are used, to perform periodic maintenance actions. Audits check and make the appropri-
ate corrections according to the database status and the detection/correction capability of the audit itself. How to tune the frequency of such checks in order to optimise system performance becomes another important aspect of the problem. This paper aims to give a contribution exactly on this last point.

In order to provide an analysis and evaluation support to help the on-line monitoring of data structures, the goal of our work consists in the definition of a methodology to model and evaluate the relevant dependability attributes of scheduled audit strategies. Contrasting, but correlated, issues have to be coped with; namely: high reliability/availability calls for frequent, deep-checking audits, while good performance in terms of accomplished services suffers from the execution power devoted to audits. We follow an analytical approach, essentially based on Deterministic and Stochastic Petri Nets (DSPN) [1, 7]. Analytical models, which capture the behaviour of the database in presence of scheduled maintenance, are defined and evaluated, in terms of identified dependability and performance measures. A sensitivity analysis with respect to the most affecting internal and external parameters is also performed on a case study, which helps in devising appropriate settings for the order and frequencies of audits to optimise selected performance indicators.

The rest of the paper is organised as follows. Section 2 presents the main characteristics of the target system and of the available audit policies. Section 3 introduces our approach to audit tuning. Section 4 discusses the identified figures of merit the assumptions made, and the basic sub-model elements used to analyse the behaviour of the database and of the audits. In section 5, a case study is set up and quantitatively evaluated to illustrate the utility of our approach; then conclusions are in Section 6.

2 System Context

We target telephone communication systems, which include a database subsystem, storing system-related as well as clients-related information, and providing basic services to the application process, such as read, write and search operations. Data concerning the status, the access rights and features available to the users, and routing information for dispatch calls are all examples of data contained in the database. The database is subject to corruption determined by a variety of hardware and/or software faults, such as internal bugs and transient hardware faults. The occurrences of such faults have the potential of yielding to service unavailability. Because of the central role played by such database in assuring a correct service to clients, means to pursue the integrity/correctness of data have to be carried out.

With the term data audit it is commonly indicated a broad range of techniques to detect errors and recover from them. The kind of checks performed on the data to test its correctness highly depends on the specific application at hand, on the system components, and environmental conditions which determine the expected fault model. Both commercial off-the-shelf and proprietary database systems are generally equipped with utilities to perform data audits, such as in [3, 4, 8]. For the purpose of our study, we assume that a set of audit procedures to cope with data corruption are provided, each characterised by a cost (in execution time) and coverage (as a measure of its ability to detect and/or correct wrong data). From the point of view of coverage, we distinguish between partial audits, characterised by a coverage lower than 1, and complete audit, which performs complete checks and recovery such that, after its
execution, the system can be considered as good as a new one. The considered audits are activated at pre-determined time intervals, in accordance with a maintenance strategy performed by an audit manager. In fact, an audit manager selects the part of the database to check/recover, the detection/recovery scheme to apply, and the frequency with which each check/recovery operation has to be performed. The audit manager is therefore responsible for applying the maintenance strategy to cope with database corruption and therefore preventing system unavailability. To set up an appropriate maintenance strategy, the audit manager would need some support, which helps it in evaluating the efficacy of applying different combinations of the available audit operations. In this work, we focus on such evaluation component (strategy evaluator), by developing a methodology to proper tuning of audit operations. In Fig. 1, the logical structure of the database subsystem and of the involved components is shown.

![Logical overview of the database subsystem](image)

Records of the database tables also include fields that are used to reference records belonging to other tables. Such reference fields (pointers) have a dynamic content. Whenever a call is set up, a set of linked records is inserted in the database; these records store all the data relevant for the establishment and management of the ongoing calls. Records allocated to store the information on a specific call are released when a call ends. The specific set of relations that identify the linked structure of the database defines the dependency scheme.

A pointer may fail in two ways: out of range, i.e., its value incorrectly assumes the value of a memory location outside the database tables, or in range, when it wrongly points to a location memory inside the tables space. The latter kind of fault shows more dangerous in the general case, since a record belonging to another dependency is erroneously deallocated; we therefore say that an in range fault generates a catastrophic failure, while an out of range fault results into a benign failure. In addition, although the single out of range fault is not catastrophic, its repeated occurrence (above a pre-fixed threshold) leads to a catastrophic failure. After a catastrophic failure, the system stops working.

In this work, we concentrate on maintenance policies for enhancing pointer correctness, which is undoubtless very critical for the application correctness; however, our approach is general methodology which can be easily adapted to take into account different specific database information.
3 A Methodology to Fine-Tuning of Audit Operations

Our goal is to identify a methodology to model and evaluate the relevant dependability attributes of scheduled audit strategies in order to derive optimal maintenance solutions. The main aspects of such a methodology are:

1. the representation of basic elements of the system and the ways to achieve composition of them;
2. the behaviour of the system components under fault conditions and under audit operations to restore a correct state;
3. the representation of failure conditions for the entire system;
4. the interleaving of audits with on-going applications and their relationships;
5. the effects of (combinations of) basic audit operations on relevant indicators for the system performance, in accordance with application requirements.

Our approach is based on Deterministic and Stochastic Petri Nets (DSPN). Specifically, in accordance with the points listed above, we defined general models which capture the behaviour of the database and of the maintenance policy checking it, to be easily adapted to specific implementations of databases and audit actions. The defined models allow investigating on the most relevant aspects in such system, related to both the integrity of the database and the overhead caused by the audit activities.

For the analysis purpose, the basic elements of the database are the pointer fields of the tables. In order to compact the basic information, one can represent in the same model structure the pointers belonging to database tables which: i) have the same failure rate; and ii) share the same audit operations, applied at the same frequency. We call the tables whose pointer fields share such characteristics as homogeneous set. Such compactness process has to be carefully performed in accordance with the set of maintenance policies to be analysed.

To represent the process of generation of pointers and of their next deletion at the end of the user call, one needs to model also the applications working on the database. This way, the events of system failure caused by erroneous pointers in dependencies at the moment of the end of a call are also captured.

Finally, the complete maintenance strategy has to be modelled, in the form of alternation of pure operational phases with others where applications and audits run concurrently.

The presentation of such general models, as well as the interactions among them, follows in the next section.

4 Modeling of Maintenance Policies

Before presenting the models, the relevant figures of merit defined for the analysis purpose and the assumptions made in our study are described.

In performing the system analysis and evaluation, we consider that the system works through missions of predefined duration [1]. To our purpose, two measures have been identified as the most sensible indicators, and the developed models have been tailored to them.

1. The reliability that should be placed on the database correctness, expressing the continuity of service delivered with respect to system specifications [5]. Actually,
to better appreciate the effect of maintenance, we will evaluate the unreliability, as a measure of the probability of not surviving a mission of a pre-fixed duration.

2. A performability measure [6], which shows appropriate to evaluate whether a certain maintenance strategy is "better" than another. Necessary to performability is the definition of a reward model; we use here, by way of example, a simple additive reward model that fits our mission-oriented systems. We assume that a gain $G_1$ is accumulated for each unit of time the system spends while performing operational phases, and a value $G_2$ is earned for each unit of time while audit operations are in execution, with $G_1 > G_2$. Finally, a penalty $P$ is paid in the case of failure, again for each time unit from the failure occurrence to the end of the mission.

The models and analysis have been developed under the following assumptions:

1. pointers corrupt with an exponential rate $\lambda_c$. Pointer faults occur independently from each other, so the corruption rate for a dependence is the sum of the corruption rates of each pointer involved in that dependence;

2. audit operations and applications share the same processor(s); when audits are in execution, a reduced number of user calls can be satisfied. The entity of such reduction, being related to audit operations, may vary significantly;

3. audit operations are characterised by a coverage $c$, indicating the audit's probability of successful detection/correction. Intuitively, the higher is $c$, the more complex (and time consuming) is the corresponding audit;

4. according to the kinds of pointer failure (i.e., in range or out of range), catastrophic or benign failures are possible, as already discussed in Section 2;

5. each active user call involves an element (record) in each database table.

4.1 The Models

Exploiting the multiple-phased structure of our problem, we developed separate models to represent a) the behaviour of the system through the alternation of operational and audit phases, and b) the failure/recovery process of the system components.

Fig. 2 shows the model of a generic maintenance strategy. It represents the alternation of a (variable) number of operational phases ($Op_1$, ..., $Op_n$) and audit phases ($Ma_1$, ..., $Ma_n$), determining a maintenance cycle, which is then cyclically re-executed. Only one token circulates in the net. The deterministic transitions $TOp_1$, ..., $TOp_n$ model the duration of operational phases, while the deterministic transitions $TMa_1$, ..., $TMa_n$ model the duration of the corresponding audit phase. The places $S_1$, ..., $S_n$ and the instantaneous transitions $TS_1$, ..., $TS_n$ allow to complete the recovery action in the homogeneous sets (described later) before a new phase starts.

![Fig. 2. Model of the maintenance strategy](image-url)
The main elements of the application sub-net, shown in Fig. 3 (a), are:

- The place *Call_active* contains the number of the on-going calls.
- The place *Corrupted* contains the number of *out of range* corruptions of a dependence (*benign failures*); one token in the place *Failed* represents the *catastrophic failure* of the system.
- The instantaneous transition *T_active* allows updating the number of tokens in the *homogeneous set*: whenever a call is set-up, represented by token moving from *Call* to *Call_active*, a token is added in the place *Table* of each homogeneous set.
- The exponential transition *T_idle* represents the duration of a call. When the system is in an operational phase, that transition fires with rate \( \mu \); during an audit phase the rate is \( x*\mu \), where \( 0 \leq x \leq 1 \) accounts for the percentage of the power processing lost during an audit phase with respect to an operational one.
- The instantaneous transitions *I_to_S*, *I_to_C*, and *I_to_F* model the behaviour of the database when a call ends. The choice of which of them fires depends on the marking of the places *actived* and *failed1* (out of range) or *failed2* (in range) in the representation of a homogeneous set sub-net (see Fig. 3(b)).

![Diagram](image)

**Fig. 3.** The application model (a) and the model of a homogeneous set (b)

Fig. 3 (b) shows the model of a *homogeneous set*, i.e., of the pointers belonging to database tables having the same failure rate and subject to the same audits, with the same frequency. The sub-nets of the application and of the homogeneous set have to be connected together, since pointers are created and deleted by user calls. The meaning of the main elements in Fig. 3 (b) is:

- The firing of the exponential transitions *Tcor* models a pointer corruption. The instantaneous transitions *Tout* and *Tin* move a token in the places *Out* and *In* respectively to distinguish if a given pointer is corrupted *out of range* or *in range.*
During a maintenance phase transitions \textit{rec\_out} and \textit{rec\_in} are enabled according to the audit specifications.

The instantaneous transitions \textit{no\_ok\_out}, \textit{ok\_out}, \textit{no\_ok\_in}, \textit{ok\_in} model the recovery actions performed at the end of audit phases. They are enabled when there is a token in the places $Sn$ of the maintenance submodel. The success or failure of a recovery is determined by the coverage $c$ of the applied audit.

When a call ends, a token (a pointer) will leave the homogeneous set sub-net. In a probabilistic way and on the basis of the marking of the places \textit{failed1}, \textit{failed2}, and \textit{actived} the decision is made on whether the dependence associated with a call is corrupted (out of range or in range) or not. The instantaneous transitions \textit{I\_to\_S}, \textit{I\_to\_C}, and \textit{I\_to\_F} of the application sub-net (see Fig. 3 (a)) operate such choice.

The instantaneous transitions \textit{to\_I}, \textit{to\_I1}, \textit{to\_I2}, \textit{to\_I3}, and \textit{to\_I4} are enabled when transition \textit{T\_Idle} of the application submodel fires and a token is moved in the place \textit{PutOut}.

The instantaneous transitions \textit{flush\_actived}, \textit{flush\_failed1}, and \textit{flush\_failed2} fire when there are no tokens in the place \textit{Idle} and after the instantaneous transitions \textit{I\_to\_S}, \textit{I\_to\_C} and \textit{I\_to\_F} of the application sub-net.

From the DSPN models, the measures we are interested in are derived as follows:

- The \textit{Unreliability} is the probability of having one token in the place \textit{Failed} (in the application model) or a given number of tokens in the place \textit{Corrupted}.
- The \textit{Performability} is evaluated with the following formula:
  \begin{equation}
  G_1^* \{\text{Operational time while the system works properly}\} + G_2^* \{\text{Audit time while the system works properly}\} - P^* \{\text{Time while the system is failed}\}.
  \end{equation}

5 A Case Study

To illustrate the usefulness of our approach and to give the reader an idea of the relevance of our analysis, a case study is set-up and evaluated.

We consider a database supporting a hypothetical telephone system, to which both partial and total audits are periodically applied. The defined maintenance strategy consists in alternating partial checks on different sets of dynamic data (pointers) with operational phases for a certain number of times, after which a complete audit is executed which resets the database status to the initial conditions. We are interested in evaluating the unreliability and performability between two complete audits; it is then straightforward to make forecasts on the system for any chosen interval of time.

By applying our methodology and composing the model elements defined in the previous section, the model instance for our case study is derived, as sketched in Fig. 4. The upper part of the model represents the maintenance strategy, which encompasses two operational phases interleaved with two executions of the some partial audit on two non-disjoint sets of data. Therefore, three homogeneous sets (A, B and C) are defined in the lower part of the model. The relationships with the application model are shown in the right side of the Fig. 4.
5.1 Numerical Evaluation

The derived models are solved by the DEEM tool [2], which provides an analytical transient solver. DEEM (DEpendability Evaluation of Multiple phased systems) is a tool for dependability modelling and evaluation, specifically tailored for multiple phased systems and therefore very suitable to be used in our context.

The variable parameters in our numerical evaluations are: i) the pointer corruption rate $\lambda_c$, which varies from $5 \times 10^{-7}$ to $5 \times 10^{-8}$ per seconds; ii) the duration of an operational phase, Top, which ranges from 60 to 300 seconds; iii) the coverage factor of partial audits, from 0.8 to 0.999; iv) the parameter P (penalty) of the reward structure.

The other involved parameters have been kept fixed; among them: the time interval between two complete audits has been set to 2 hours; the maximum number of user calls concurrently active is 100; the call termination rate is $3.33 \times 10^{-3}$ per seconds; the number of benign failures necessary to determine a catastrophic system failure is 5; the parameters $G_1$ and $G_2$ of the reward structure.

Fig. 5(a) shows the performability as a function of the duration of the operational phase, for different values of the penalty associated to the failure condition of the system. For the chosen setting, it can be observed a noticeable influence of such penalty factor P on the resulting performability. When P is just a few times the value of $G_1$ (i.e. the gain in case the system is fully and correctly operating), increasing Top brings benefits to the performability. This means that in such a case, the main contribution to the performability is given by the reward accumulated over operational phases. However, for P from 200 to 300, an initial improvement can be observed,
which is then lost, although the performability degradation is not dramatic. When P is two order of magnitudes higher than G1, the cost of a failure is so big that lengthening Top (which implies a higher risk of failure) results in a performability detriment.

Fig. 5(b) shows the performability keeping fixed the reward structure and at varying values of the coverage of the audit procedure and the length of the operational phase. Two effects can be immediately noticed. First, as expected, the performability improves with growing values of the coverage. Second, it can be observed a “bell shape” of the curves: the performability grows at growing values of the duration of the operational phase till a maximum value of Top after which the trend inverts.

**Fig. 5.** Performability at varying of Top and Penalty (a) and Coverage (b) respectively

In fact, the higher reward obtained during a longer operational phase is at first the dominant factor in determining the performability, but lengthening Top also means exposing the system to a higher risk of failure, and the penalty P to be paid in such a case becomes the most influencing parameter in the second part of the Fig. 5(b).

**Fig. 6.** Performability at varying Top and \(\lambda_c\) (a) and Unreliability (b) at varying \(c\) and Top

Fig. 6(a) completes the analysis of the performability, at varying values of Top and for three different values of the pointer failure rate. The impact of \(\lambda_c\) on the performability is noteworthy, and behaviour similar to that in Fig. 5(a) is observed. Fig. 6(b) shows the behaviour of the unreliability at varying values of the coverage and for several values of Top. Of course, the unreliability improves at increasing both the audits frequency (i.e., small Top) and the coverage of the audits. It can be noted that
same values of the unreliability can be obtained by adopting audits with a higher coverage or applying more frequently audits with a lower coverage.

The analyses just discussed give a useful indication about the tuning of the major parameters involved in the database system. The optimal trade-off between the frequency of the audits and the investment to improve the coverage of the audits can be found, to match the best performability and dependability constraints.

6 Conclusions

This paper has focused on maintenance of dynamic database data in a communication system. To achieve a good trade-off in terms of overhead and efficacy of the maintenance, it is necessary to properly choose which audit operations are to be applied and how frequently they should be used.

We proposed a modular methodology to model and evaluate the relevant dependability attributes of scheduled audit strategies. Our approach is based on Deterministic and Stochastic Petri Nets (DSPN) and on the DEEM tool. Despite our proposed approach needs further work for being assessed, nevertheless we have identified several relevant characteristics specific to this class of systems.

The major impact of this study is the definition of a general model for the evaluation of the effectiveness of the audit strategies. Paramount criteria for our work have been the extensibility and flexibility in composing the audit strategies. Of course, in order for our models to be really useful for the selection of proper order and frequencies of audit operations, input parameters such as cost and coverage of the checks and failure data are necessary and should be provided. Investigations to assess the merits of our approach towards the incremental structure of audit methods are planned as the next step. Also, extensions of the case study to include the comparison of the effectiveness/benefits derived from applying different combinations of audits (i.e., different maintenance strategies) constitute interesting evolution to this work.

References

## Author Index

<table>
<thead>
<tr>
<th>Author</th>
<th>Page</th>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arlat, J.</td>
<td>130</td>
<td>Laible, U.</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Le Guédart, Y.</td>
<td>130</td>
</tr>
<tr>
<td>Betous-Almeida, C.</td>
<td>227</td>
<td>Mäckel, O.</td>
<td>92</td>
</tr>
<tr>
<td>Bitsch, F.</td>
<td>176</td>
<td>Majzik, I.</td>
<td>46</td>
</tr>
<tr>
<td>Bondavalli, A.</td>
<td>238</td>
<td>May, J.H.R.</td>
<td>156</td>
</tr>
<tr>
<td>Boyer, T.</td>
<td>130</td>
<td>Mazuet, C.</td>
<td>215</td>
</tr>
<tr>
<td>Bürger, T.</td>
<td>75</td>
<td>Naedele, M.</td>
<td>25</td>
</tr>
<tr>
<td>Carter, R.B.</td>
<td>156</td>
<td>Nordland, O.</td>
<td>56</td>
</tr>
<tr>
<td>Caspi, P.</td>
<td>215</td>
<td>Pataricza, A.</td>
<td>46</td>
</tr>
<tr>
<td>Chohra, A.</td>
<td>238</td>
<td>Porcarelli, S.</td>
<td>238</td>
</tr>
<tr>
<td>Cichocki, T.</td>
<td>190</td>
<td>Pritschow, G.</td>
<td>75</td>
</tr>
<tr>
<td>Di Giandomenico, F.</td>
<td>238</td>
<td>Pasquini, A.</td>
<td>100</td>
</tr>
<tr>
<td>Dzung, D.</td>
<td>25</td>
<td>Pritschow, G.</td>
<td>75</td>
</tr>
<tr>
<td>Faller, R.</td>
<td>200</td>
<td>Patschke, S.</td>
<td>238</td>
</tr>
<tr>
<td>Fankhauser, H.R.</td>
<td>66</td>
<td>Patschke, S.</td>
<td>238</td>
</tr>
<tr>
<td>Fox, J.</td>
<td>1</td>
<td>Redmill, F.</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reumann, C.</td>
<td>166</td>
</tr>
<tr>
<td>Gallardo, J.</td>
<td>156</td>
<td>Reynaud Paligot, N.</td>
<td>215</td>
</tr>
<tr>
<td>Gawkowski, P.</td>
<td>109</td>
<td>Rizzo, A.</td>
<td>100</td>
</tr>
<tr>
<td>Gingerl, M.</td>
<td>166</td>
<td>Rushby, J.</td>
<td>86</td>
</tr>
<tr>
<td>Górski, J.</td>
<td>190</td>
<td>Save, L.</td>
<td>100</td>
</tr>
<tr>
<td>Gran, B.A.</td>
<td>14, 35</td>
<td>Sosnowski, J.</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stanimirov, M.</td>
<td>25</td>
</tr>
<tr>
<td>Helminen, A.</td>
<td>35</td>
<td>Vakalis, I.</td>
<td>145</td>
</tr>
<tr>
<td>Hoever, G.</td>
<td>92</td>
<td>Vakalis, I.</td>
<td>145</td>
</tr>
<tr>
<td>Hughes, G.</td>
<td>156</td>
<td>Vakalis, I.</td>
<td>145</td>
</tr>
<tr>
<td>John, A.D.</td>
<td>156</td>
<td>Winther, R.</td>
<td>14</td>
</tr>
<tr>
<td>Johnsen, O.-A.</td>
<td>14</td>
<td>Winther, R.</td>
<td>14</td>
</tr>
<tr>
<td>Kaâniche, M.</td>
<td>130</td>
<td>Zoffmann, G.</td>
<td>166</td>
</tr>
<tr>
<td>Kanoun, K.</td>
<td>227</td>
<td>Zoffmann, G.</td>
<td>166</td>
</tr>
<tr>
<td>Kuball, S.</td>
<td>156</td>
<td>Zoffmann, G.</td>
<td>166</td>
</tr>
</tbody>
</table>