Rune Winther
Bjørn Axel Gran
Gustav Dahll (Eds.)

LNCS 3688

Computer Safety, Reliability, and Security

24th International Conference, SAFECOMP 2005
Fredrikstad, Norway, September 2005
Proceedings

Springer

Rune Winther Bjørn Axel Gran
Gustav Dahll (Eds.)

Computer Safety, Reliability, and Security

24th International Conference, SAFECOMP 2005
Fredrikstad, Norway, September 28-30, 2005
Proceedings

Springer

Volume Editors

Rune Winther
Østfold University College
Faculty of Computer Sciences
1757 Halden, Norway
E-mail: rune.winther@hiof.no

Bjørn Axel Gran
Gustav Dahll
Institute for Energy Technology
Software Engineering Laboratory
1761 Halden, Norway
E-mail: bjorn.axel.gran@hrp.no; g.dahll@halden.net

Library of Congress Control Number: 2005932842

CR Subject Classification (1998): D.1-4, E.4, C.3, F.3, K.6.5

ISSN	0302-9743
ISBN-10	3-540-29200-4 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-29200-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11563228 06/3142 5 4 3 2 1 0

Preface

Welcome to SAFECOMP 2005, held in Fredrikstad, Norway. Since its establishment SAFECOMP, the series of conferences on Computer Safety, Reliability and Security, has contributed to the progress of the state of the art in dependable applications of computer systems. SAFECOMP provides ample opportunity to exchange insights and experiences in emerging methods and practical experience across the borders of different disciplines. Previous SAFECOMPs have year after year registered new multidisciplinary trends on dependability of computer-based systems.

SAFECOMP 2005 focused on dependability of critical computer applications and was a platform for knowledge and technology transfer between academia, industry and research institutions. Papers were invited on all aspects of dependability and survivability of critical computer-based systems in various branches and infrastructures. Due to the increasing awareness and importance of security issues of critical computer-based systems, SAFECOMP 2005 emphasized work in this area. Nowadays practical experience points out the need for multidisciplinary approaches to deal with the nature of critical complex settings.

The SAFECOMP 2005 program consisted of 30 papers selected from 84 submissions. The 30 papers represented scientists from 14 different countries acknowledging the world-wide interest of SAFECOMP and the addressed topics. The SAFECOMP program was supplemented by keynote talks enhancing the technical and scientific merit of the conference, a number of co-located activities, meetings and tutorials, and a technical visit to the research environment in Halden which organized the conference.

We would like to thank the International Program Committee, the external reviewers, the keynote speakers, and the authors for their work in support of SAFECOMP 2005. We would also like to thank the conference staff at the Institute of Energy Technology and Østfold University College. We really enjoyed the work, and we hope you appreciated the care we put into organizing the conference. Finally, we would like to extend to you the invitation to attend and contribute to SAFECOMP 2006 in Gdansk, Poland (www.safecomp.org).

July 2005

Gustav Dahll
Bjørn Axel Gran
Rune Winther

Organization

SAFECOMP 2005 Organization

General Chair

Gustav Dahll, Norway

Program Co-chairs

Bjørn Axel Gran, Norway
Rune Winther, Norway

EWICS Chair

Udo Voges, Germany

Organizing Committee

Rune Fredriksen, Norway
Siv-Hilde Houmb, Norway
Monica Kristiansen, Norway
John Petter Kvalvik, Norway
Jørn L. Pettersen, Norway
Atoosa P.-J. Thunem, Norway
Bjørn Axel Gran, Norway
Rune Winther, Norway

IFE Institute for Energy Technology

Høgskolen i Østfold

International Program Committee

External Reviewers

Sponsoring Organizations

Scientific Sponsor

EWICS TC7 Reliability Safety Security

Scientific Co-sponsors

The Research Council of Norway

seibersdorf research
An enterprise of the Austrian Research Centers

NTNU
Norwegian University of
Science and Technology

telenor

SCANDINAVIAN Reliability Engineers

CORAS

OESTERREICHISCHE COMPUTER GESELLSCHAFT
AUSTRIAN COMPUTER SOCIETY

IFIP WG10.4 on Dependable Computing and Fault Tolerance
IFIP WG13.5 on Human Error, Safety and System Development

ifip the leading edge of information technology

BCS THE BRITISH COMPUTER SOCIETY

THE UNIVERSITY OF EDINBURGH

European Network of Clubs for REliability and Safety of Software

Halden IT Forum
SCSC - Safety-Critical Systems Club
SRMC - Software Reliability & Metrics Club
NONSTOPP - NOrsk Nettverk for Sikre Trygge Og Pålitelige Programmerbare Systemer

Table of Contents

CMMI RAMS Extension Based on CENELEC Railway Standard

Jose Antonio Fonseca and Jorge Rady de Almeida Júnior

Computational and Digital Systems Engineering Department,
Polytechnic School, University of São Paulo, Brazil

Abstract. Railway systems are also dependable systems and, considering their importance, it is vital to assure the application of adequate design techniques. So, this work presents a RAMS (Reliability, Availability, Maintainability and Safety) extension for CMMI SE-SW version 1.1 "Capability Maturity Model® Integration" developed by SEI (Software Engineering Institute), based on CENELEC 50126, 50128 and 50129 standards developed to normalize RAMS aspects of railway control systems in European Community. This extension is based on the inclusion of four new Process Areas into the CMMI SE-SW, increasing its actual number from 22 to 26, without changes in the CMMI model basic structure. The objective of this extension is to obtain a support tool for design process applicable to enterprises that develop railway systems and are adopting CMMI or migrating from other CMM models.

1 Introduction

This work represents an attempt to join two very important tendencies that are being verified by the maturity models use and the railway applications design. Considering the great capacity of CMM models to assist innumerous application areas, the first trend can be observed through an increase in the use of maturity models by industrial community. The focus of such models is represented by CMMI (Capability Maturity Model Integration). The second trend is composed by integration efforts to create a consensus about RAMS (Reliability, Availability, Maintainability and Safety) criteria for railways applications between the European Union members that is represented by CENELEC standards.

This work has also a very closely relationship with others efforts to incorporate new specifics aspects to CMMI, such as the job sponsored by FAA (Federal Aviation Administration) to include Safety and Security requirements in iCMM and CMMI and the task headed by Australian Government's Defense Material Organization (DMO) in the creation of +Safe, a safety extension to CMMI.

The section 2 presents a brief description about the CMMI model, while section 3 presents the mains aspects of the CENELEC Standards. Section 4 contains the proposed extension of RAMS extension for CMMI model. Finally, section 5 presents the mains conclusions of this paper.

2 The CMMI Model

The extensive use of the SW-CMM [9] (Capability Maturity Model for Software) by the organizations promoted the creation of similar models to address other areas not directly related with software development. Considering such aspect, many other models have arisen to support production systems, subcontracting areas, etc. But, all of these models were not created in order to facilitate integration among them, generating problems with their simultaneous implementation in an organization.

This fact has revealed the need of creating an integrated model, aiming a uniform view, besides the elimination of existing redundancies among the various maturity models. We can say that the CMMI is a result of a great integration work, and that it was elaborated to allow a convergence of the main existing maturity models. The CMMI structure also allows integration of new areas, which reinforces its integration capacity.

The CMMI SE/SW (Capability Maturity Model for Systems and Software Engineering) model V1.1 [5] consists of 22 Process Areas. A Process Area is a group of related practices that, when accomplished together, means that a set of important objectives were achieved, obtaining a significant improvement in such area.

All the CMMI Process Areas are common to the stage representation and the continuous representation. In the <u>stage representation</u>, the Process Areas are organized through <u>maturity levels</u>. Considering one level, all of its Process Areas are in the same maturity level. In the <u>continuous representation</u>, the maturity of a Process Area is called <u>capability level</u> and each Process Area can be in any of the six capability levels existents, independently of any other Process Area.

Thus, the name “maturity level” refers to a pre-defined group of Process Areas, which are in the same maturity level, whereas “capability level” refers only to an individual Process Area.

The continuous representation allows that one organization can choose the more adequate improvement sequence to its business goals, making possible a reduction of the risk areas.

The stage representation also offers a series of improvements, starting from basic management practices and going through a predefined plan of successive levels where each level is the basis for the next one.

To completely satisfy a Process Area, both <u>generic and specific goals</u> must be accomplished. Specific goals are applied to a Process Area and refer to single characteristics, which describe what has to be done to satisfy a Process Area.

The <u>specific goals</u> are supported by specific practices which are activities considered important to achieve a specific goal. The specific practices describe the activities, which must be accomplished in order to reach a specific goal of a Process Area.

<u>Generic goals</u> are called “generic” because a single goal can appear in multiple Process Areas. Considering the staged representation, every Process Area has a single specific goal. Generic goals are supported by common practices.

The CMMI continuous representation allows one organization to keep its capacity on the improvement of a single Process Area, or on multiple specific Process Areas. Each Process Area has its own specific goals associated similarly to the staged representation. Each capability level (from 0 to 5) has a common goal and many common practices.

The staged representation does not have requirements for the first maturity level; whereas, in the continuous representation there are specific and generic goals to be accomplished in order to achieve capability level 1. This has increased the granularity of the capability (process maturity), in such way that the organizations show early progress. This can be important in organizations that are under pressure to present immediate results.

The 22 Process Areas are divided into four categories, according to figure 1. In the activity of selecting a Process Area or a single category, an organization can focus its improvement efforts in such area. Each one of the 22 Process Areas can be characterized individually by the CMMI as having a maturity level from 0 through 5, as follows:

Capability Level 0 - Incomplete
An incomplete process is a partially accomplished or a non-accomplished process, that is, at least one of the specific goals of the Process Area is not achieved.

Capability Level 1 – Executed
At this level, processes achieve the specific goals of the correspondent Process Area. The process supports the necessary work to generate the required products from the inputs, which are correctly identified during the process. The difference between an incomplete process and an executed process is that an executed process achieves all the specific goals of the Process Area.

Capability Level 2 – Managed
A managed process consists in an executed process (capacity level 1), which is also planned and executed, according to a plan, which embraces qualified people, adequate resources and appropriate participants. The process is monitored, controlled, revised and evaluated according to its process description adherence and it can be instantiated to a design, group or organizational function. The process management comprises the Process Area institutionalization and the accomplishment of other specific objectives defined for the process, such as cost, time schedule and quality goals.

Capability Level 3 – Defined
A defined process is a managed process (capacity level 2), which includes a group of default processes according to the organization objectives, its metrics, and other information on process improvement.

Capability Level 4 – Quantitatively Managed
A quantitatively managed process is a defined process (capacity level 3), which is controlled through the use of statistics and other quantitative techniques. The quantitative objectives of quality and process performance are established and used as a criterion in the process management. The quality and process performance are transformed into statistics expressions and managed through the process lifecycle.

Capability Level 5 – Optimized
An optimized process is a quantitatively managed process (capacity level 4), which is modified and adapted to achieve the business and relevant goals in a specific moment. An optimized process is focused on the continuous improvement of the process performance through the use of technological improvement and innovative technologies.

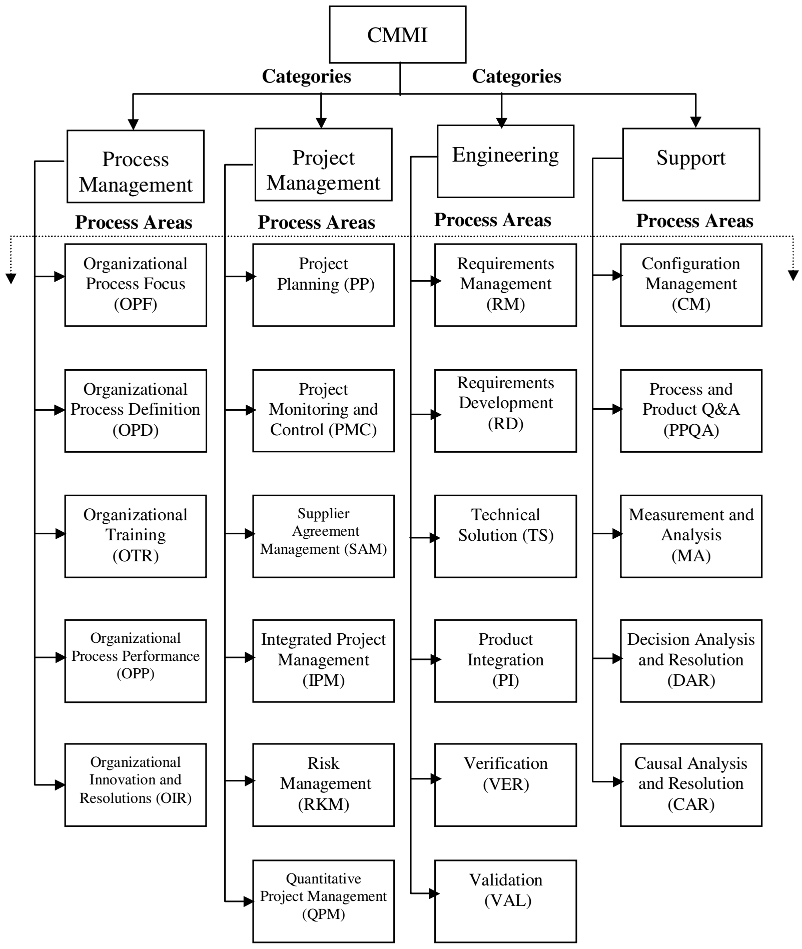

CMMI
Categories
Categories
Process Management
Project Management
Engineering
Support
Process Areas
Process Areas
Process Areas
Process Areas
Organizational Process Focus (OPF)
Organizational Process Definition (OPD)
Organizational Training (OTR)
Organizational Process Performance (OPP)
Organizational Innovation and Resolutions (OIR)
Project Planning (PP)
Project Monitoring and Control (PMC)
Supplier Agreement Management (SAM)
Integrated Project Management (IPM)
Risk Management (RKM)
Quantitative Project Management (QPM)
Requirements Management (RM)
Requirements Development (RD)
Technical Solution (TS)
Product Integration (PI)
Verification (VER)
Validation (VAL)
Configuration Management (CM)
Process and Product Q&A (PPQA)
Measurement and Analysis (MA)
Decision Analysis and Resolution (DAR)
Causal Analysis and Resolution (CAR)

Fig. 1. CMMI Continuous Representation Process Areas

3 The CENELEC Standards

Since the first steps towards a single market of railway transport services in the European Union, it became evident the existence of different regulations in the safety issue.

The main reason for this situation can be explained by the fact that local national operators, which have all the responsibility for the systems operation inside their territories, perform the railway transport management of these countries. However, considering the increasingly integration of the European railway systems, the safety aspect should be considered in the most general ambit of the European Union [1].

At present, the railway industry is observing a process of developing appropriate safety standards that can control the new devices created by the technology development, seeking to ensure the adequate safety level for the systems. Railway suppliers are looking for standards that can aid them to show evidences about the safety of their products. Railway owner/operators are also insecure about what they can expect from suppliers, in order to make them feel more comfortable in accepting a product [6], [7].

The European Union, through the European Committee for Electrotechnical Standardization (CENELEC), has been developing standards for the safety issues regulation for railway application [1]. All the European Union members desire to turn possible the interoperability among the several existing systems, through the adoption of such standards.

The main focus of the CENELEC standards is composed by a systematic hazard identification followed by a risk reduction at an acceptable level. Considering the necessary risk reduction, CENELEC standards “recommend” techniques and methods that permit to demonstrate that the required RAMS targets levels are satisfied.

CENELEC standards consider the critical systems design according to the focus shown in the figure 2. Some of the activities presented in that figure are described in a simplified way.

The philosophy adopted by CENELEC Standards is based in systematic hazard identification [8]. For each identified hazard there must be, at least, one requirement to eliminate or mitigate the risk associated to that hazard.

Since the initial railway design phases, all RAMS related activities are made in agreement with CENELEC standards, considering also the SIL (Safety Integrity Level) required for the application, as defined by the operator [6] or by the supplier, according to a National Safety Regulatory Office. This phase product is a Safety Plan report that defines all design activities, including audits, responsibilities, roles and design schedules.

The Hazard Identification and risk reduction are activities performed during all the design lifecycle. In early phases, it is made a Preliminary Hazard Analyses (PHA), whose objective is to define a basic set of safety requirements that must be detailed in posterior phases. This activity must consider the Hazard Log contents. This Hazard Log is constantly updated along the system design development with new identified hazards.

The requirements identified in previous phases are refined in the design phase, where the solutions needed to satisfy the identified requirements are developed. The CENELEC Standards describes the main activities needed to demonstrate that the RAMS principles were correctly applied during each development phase, always considering the selected SIL [2], [3], [4].

For CENELEC, the SIL define the depth which the design should be analyzed, the necessary evidences to demonstrate that all specified requirements are satisfied and define the hierarchical constrains and the required personal skills to the design development.

Audits are performed during system development, whose function is to verify the compliance with directives contained in Plans for Quality Assurance, for Configuration Management, for Safety and others. A report is generated for each audit aspect, in order to demonstrate the design status.

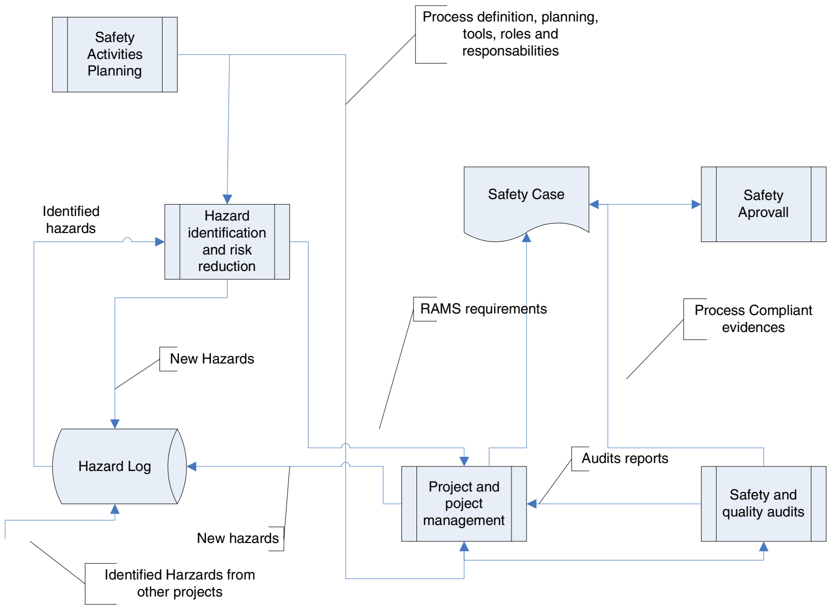

Safety Activities Planning
Process definition, planning, tools, roles and responsabilities
Safety Case
Safety Aprovall
Identified hazards
Hazard identification and risk reduction
RAMS requirements
Process Compliant evidences
New Hazards
Hazard Log
Audits reports
Project and poject management
Safety and quality audits
New hazards
Identified Harzards from other projects

Fig. 2. CENELEC Basic Process

The set of evidences collected during system development and design deliverables are used to elaborate a Safety Case, also known as Technical Safety Report (TSR). A TSR is a dossier that demonstrates that RAMS requirements were correctly captured, satisfied and traceable.

4 The RAMS Extension for CMMI

As previously mentioned, both CMMI model and CENELEC standards propose concept unification in their application areas. But, organizations that are developing safety-critical systems need to consider the concepts established in those two frameworks in an integrated way, which is very difficult, considering that the their structures does not consider such kind of integration.

The main problem in using both CMMI and CENELEC in the same application is that CMMI does not attend the RAMS aspects. This fact makes necessary to apply another specific standard, in order to supply such RAMS aspects. However, CENELEC does not provide, itself, an evaluation system and a guide to its implementation, like CMMI. On the other hand, the simultaneous application of the CMMI and CENELEC standards, in a separate way, without a previous harmonization, can generate problems such overlaps, different interpretations and misunderstandings.

Thus, the proposal of this work is to make the unification of these two frameworks into a single structure, flexible enough to support the necessary adaptations.

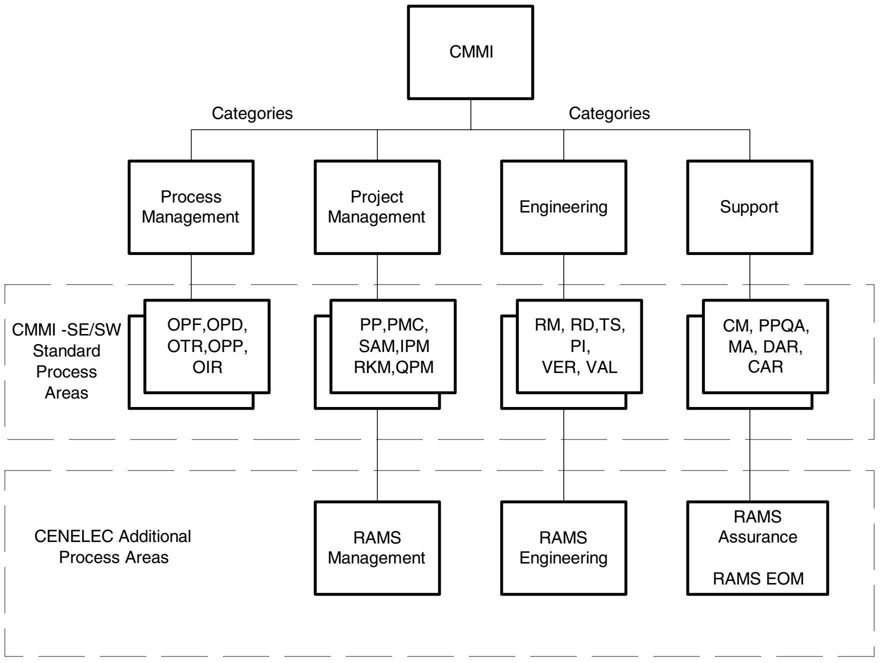


Fig. 3. RAMS Extension Structure

This paper proposes the addition of RAMS into the CMMI structure, without modifying its fundamental structure, which is widely accepted. So, the CENELEC standard arrangement is modified to attend the CMMI language. If we have tried the inverse solution, that is, incorporate CMMI into CENELEC standards, there would be a problem related to the great quantity of practices to be inserted into the CENELEC standards, in order to cover all the CMMI aspects. This happens because CENELEC standards are specific in their area and CMMI is a common model for systems. A second problem would be the generation of unnecessary redundancies and the need of generating a mapping between the model and the standard.

We believe that the introduction of the CENELEC concepts into CMMI will be more natural for the most organizations, reducing implementation costs and time, when considering the application of these concepts in a separate way.

Primarily the RAMS extension was implemented in CMMI SE-SW continuous representation (presented in figure 1), because we believe that continuous representation provide the necessary flexibility according to organizations needs, schedules and budget, although staged representation can also be used.

To satisfy CENELEC Standards, besides the inclusion of new RAMS Process Areas, it is necessary to implement the existent areas of CMMI. Note that these Process Areas are distributed in levels 2 and 3 of the staged representation. Therefore it is necessary to implement the level 3 for these representations to satisfy CENELEC standards. In addition, some Process Areas of CMMI, above mentioned, exceed CENELEC requirements, but this fact contribute with CENELEC process improvement.

The new Process Areas are included in CMMI SE-SW continuous representation as show in figure 3. As shown, the four new Process Areas are distributed between three CMMI categories, as follow:

Table 1. New Process Areas

CMMI Category	Process Areas
Project Management	RAMS Management
Engineering	RAMS Engineering
Support	RAMS Assurance RAMS Environment Organization and Maintenance

The main objectives of the new Process Areas and their respective Specific Goals (SG) are:

RAMS Management

Purpose
The aim of this Process Area is to monitor the product development process, checking if the RAMS activities are performed as planned and tracking the design evolution.

Related Process Areas
There are many processes areas related with RAMS Management. These areas and their respective connections with RAMS Management are:

- Requirements Development: acquisition of information about developing requirements that define the product and product components;
- Requirements Management: acquisition of information about managing requirements needed for planning and re-planning;
- Technical Solution: acquisition of information about transforming requirements into product and product component solutions;
- Organizational Process Definition: acquisition of information about the design lifecycle and basic guidelines; and
- RAMS Environment Organization and Maintenance: identification of organization requirements, knowledge and skills.

Practice-to-Goal Relationship Table

SG 1. Develop a RAMS Plan

SP1.1-1	Establish Validation Strategy
SP1.2-1	Establish RAMS Organization, Roles and Responsibilities
SP1.3-1	Establish RAMS Lifecycle to the Design

SP1.4-1 Establish Audits and Assessments Points
SP1.5-1 Plan for Data Management
SP1.6-1 Plan for Resources
SP1.7-1 Plan for Needed Knowledge and Skills
SP1.8-1 Plan Stakeholder Involvement
SP1.9-1 Plan Safety Reviews
SP1.10-1Establish the RAMS Plan

SG 2. Obtain Commitment to the Plans
SP 2.1-1 Review Plans that Affect the Design
SP 2.2-1 Reconcile Work and Resource Levels
SP 2.3-1 Obtain Plans Commitments

SG 3. Develop an Installation and Commissioning Plan
SP3.1-1 Establish Installation and Commissioning Strategy
SP3.2-1 Establish Roles and Responsibilities
SP3.3.1 Plan for Resources
SP3.4.1 Plan for Needed Knowledge and Skills
SP3.5.1 Plan Stakeholder Involvement
SP3.6.1 Establish the RAMS Plan

SG 4. Monitor Safety Incidents
SP4.1-1 Monitor Safety Incidents

RAMS Engineering

Purpose
The aim of the RAMS Engineering is to define the activities that must be performed, in order to assure that the generated products have the desired and adequate RAMS levels for the application.

Related Process Areas
There are many processes areas related with RAMS Engineering. These areas and their respective connection with RAMS Engineering are:

- Requirements Development: decision on how to allocate or distribute requirements among the product components;
- Technical Solution: acquisition of more information about RAMS decisions;
- Project Planning: how design plans reflect requirements and need to be revised with changes in requirements;
- Configuration Management: obtain information about baselines and controlling changes, considering configuration issues;
- Project Monitoring and Control: activities track and control, taking appropriate corrective actions; and
- Requirements Management: get information about managing requirements.

Practice-to-Goal Relationship Table

SG 1. Identify Safety Requirements
SP1.1-1 Perform Hazard Analysis
SP1.2-1 Perform Risk Assessment
SP1.3.1 Define Risk Tolerance Criteria

SG 2. Develop Safety Requirements
SP2.1-1 Allocate Safety Requirements to Products
SP2.2-1 Apply Safety Principles to Safety Requirements
SP2.3-1 Justify Technical Safety Decisions

SG 3. Establish a Hazard Log
SP3.1-1 Establish a Hazard Log to the Design

SG 4. Identify RAM requirements
SP4.1-1 Perform Preliminary RAM Analysis
SP4.2-1 Identify External Influence over RAM Requirements

SG 5. Develop RAM Requirements
SP5.1-1 Allocate RAM Requirements to Products
SP5.2-1 Justify Technical RAM Decisions

SG 6. Demonstrate the Safety of the system
SP6.1-1 Perform Analysis about Effects of Faults
SP6.2-1 Perform Analysis about External Influences
SP6.3-1 Perform Analysis about Application Conditions
SP6.4-1 Perform Safety Qualification Tests
SP6.5-1 Develop a Technical Safety Report

RAMS Assurance

Purpose
The aim of the RAMS Assurance is to evaluate, continuously, the correct accomplishment of design lifecycle activities and the related RAMS products, which are generated by engineering activities, in order to assure the product integrity.

Related Process Areas
There are many processes areas related with RAMS Assurance. These areas and their respective connection with RAMS Assurance are:

- Project Planning: identification of the processes and the associated products that need to be evaluated;
- Verification: satisfaction of RAMS requirements;
- Process and Product Quality Assurance: audits on RAMS process and evidences;
- RAMS Management: acquisition of safety management evidences
- RAMS Engineering: acquisition of evidences about functional/technical safety
- RAMS Environment Organization and Maintenance: identification of knowledge and skills.

Practice-to-Goal Relationship Table

SG 1. Develop an Assessment Plan

SP1.1-1 Establish Assessment Strategy
SP1.2-1 Establish Roles and Responsibilities
SP1.3-1 Plan for Resources
SP1.4-1 Plan for Needed Knowledge and Skills
SP1.5-1 Plan Stakeholder Involvement
SP1.6-1 Identify Design, Personnel or Documents Dependencies
SP1.7-1 Establish the Assessment Plan

SG 2. Perform evaluations

SP2.1-1 Perform Interviews with Design Personnel
SP2.2-1 Perform Examination of Design Documents
SP2.3-1 Perform Observation of Practices, Design Activities and Conditions
SP2.4-1 Re-work of Parts of the Safety Analysis if Necessary
SP2.5.1Demonstrations Arranged at the Assessor's Request
SP2.6.1Elaborate an Assessment Report

SG 3. Develop a Safety Case

SP3.1-1 Collect evidences of Quality Management
SP3.2-1 Collect Evidences of Safety Management
SP3.3-1 Collect Evidences about Functional/Technical Safety
SP3.4-1 Develop a Safety Case

RAMS Environment Organization and Maintenance

Purpose

The aim of the RAMS Environment Organization and Maintenance is to create a suitable infrastructure to support RAMS activities, select RAMS specialists, and define roles and responsibilities.

Related Process Areas

The Process Area related with RAMS Environment and Maintenance is the Organizational Training that can be consulted for information about how to creates skills.

Practice-to-Goal Relationship Table

SG 1. Establish environment and organization to RAMS Activities

SP1.2-1 Identify Overall RAMS Organization
SP1.1-1 Identify and Create Necessary Skills
SP1.3-1 Define Roles and Responsibilities
SP1.4-1 Identify Necessary Independence for Activities
SP1.5-1 Maintain the Qualification of Environment Components
SP1.5-1 Plan for Continuity and Improvements

Final Remarks

To provide CMMI compatibility with CENELEC standard, some CMMI Process Areas need modifications. For Verification and Validation Process Areas, it is necessary to formalize Verification and Validation Plans and for the Organizational Process Definition Process Area it is necessary to adopt CENELEC lifecycle descriptions for software and system development.

5 Conclusions

This paper presented an extension for CMMI based on the CENELEC standards, having the objective of aiding organizations that develop railway systems to work in an integrated way with CMMI and CENELEC standards, enabling reduction of time and its associated costs, when compared with an implementation using those two frameworks in an isolated way.

After a careful refinement process, we believe that this work could come to be very useful to the railway applications.

However, this extension does not include all the details of CENELEC standards because it does not define, specifically, as the activities should be done. These details should be defined for the organizations in accordance with the design needs of the based on CENELEC.

In that way, the basic structure of the proposed extension is generic and could be applied to other areas just needing the inclusion of the specific practices of that new area.

In future works it will be detailed the specific practices for each specific objective and will be analyzed the compatibility between Process Areas of this work and other works such as +Safe and FAA-iCMM.

References

1. CEC – DIRECTIVE OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL, Commission of the European Communities, Brussels, 2002.
2. CENELEC126- COMITÉ EUROPÉEN DE NORMALISATION ÉLECTROTECHNIQUE. Railway applications: The specification and Demonstration of Reliability, Availability, Maintainability and Safety – BS EN50126. Brussels, CENELEC, 1999.
3. CENELEC128- COMITÉ EUROPÉEN DE NORMALISATION ÉLECTROTECHNIQUE. Railway applications: Software for railways control and protection systems – BS EN50128. Brussels, CENELEC, 2001.
4. CENELEC129- COMITÉ EUROPÉEN DE NORMALISATION ÉLECTROTECHNIQUE. Railway applications: Safety related electronic system for Signalling – BS EN50128. Brussels, CENELEC, 2003.
5. CMMI SE-SW- The Capability Maturity Model Integration, ver. 1.1, December 2001 Software Engineering Institute.
6. McNICOL M. "Signalling systems, a view to the future". Railsafe 99 26-27 jully, 1999 Sydney, Australy
7. NERA - Safety Regulation and Standards for European Railways February 2000, London.
8. SCHABE H. "The Safety Philosophy behind the CENELEC Railways Standards", 2002.
9. SW-CMM – Capability Maturity Model for Software, version 2.0,1997, software Engineering Institute.

The Importance of Single-Source Engineering of Emergency and Process Shutdown Systems

Robert Martinez and Torgeir Enkerud

ABB AS, Corporate Research Center, Bergerveien 12, PO Box 90,
NO-1375 Billingstad, Norway
`robert.martinez@no.abb.com, torgeir.enkerud@no.abb.com`

Abstract. Emergency/Process ShutDown systems (ESD/PSD) involve large numbers of signals, span many process units and have strict compliance requirements. These factors increase the burden of engineering, operation and reporting, and drive the search for techniques such as Cause & Effect Matrix (CEM). By showing input signals as matrix rows and outputs as columns, CEM provides an intuitive view of shutdown trip logic and is now common practice in industry. This popularity has contributed to problems of data duplication and transcription errors when multiple incarnations of the same CEM are used at different lifecycle stages. Process engineers, programmers, operators and safety managers each view the same CEM recreated in different formats.

The authors show how the CEM paradigm can benefit from a standardised syntax and visual representation so that all the different views of a CEM are based on the same underlying data, increasing safety and productivity throughout the lifecycle.

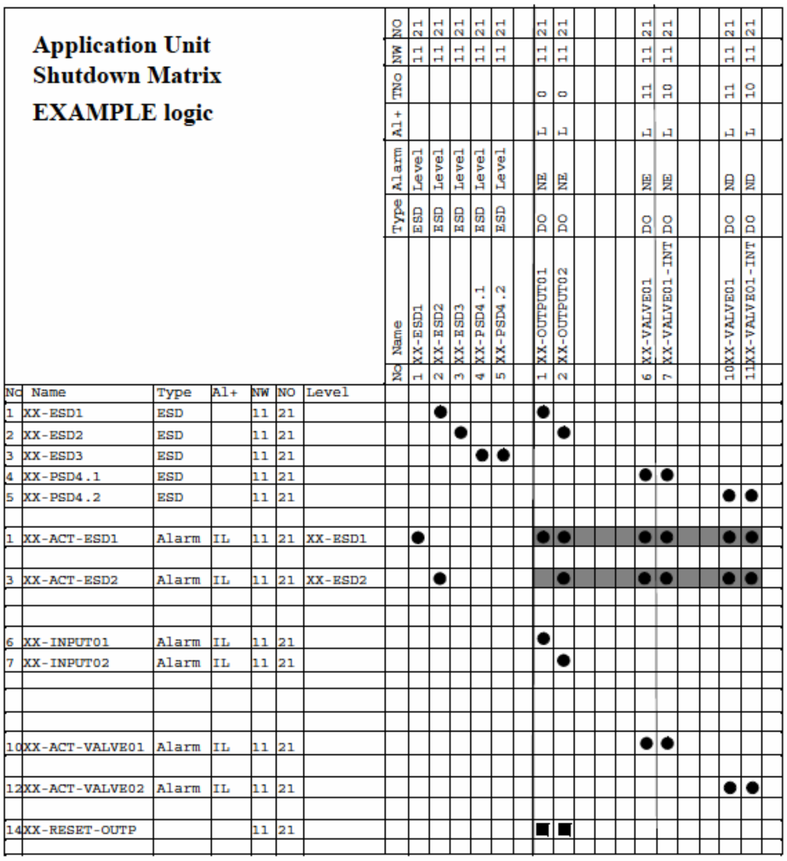
Keywords: Emergency shut down, cause and effect, single-source engineering.

1 Introduction

On the face of it, the design of ESD/PSD safety systems appears trivial. There are no complex control algorithms to design and test, no parameters to tune, no troublesome analog values to filter; just Boolean variables inputs setting Boolean outputs.

The Cause and Effect Matrix (CEM) visual paradigm, also known as Interlock Logic Diagrams, was introduced many years ago to capture this clear signal flow from cause to effect which is the hallmark of ESD/PSD systems. In its simplest form, it is a matrix of gridlines where named tags occupy rows and columns. A symbol placed at a row/column intersection cell indicates that a trip of that column effect will occur when that row cause is active. The simple CEM interface evolved to make room for shutdown levels, which allowed users to define a cascading shutdown hierarchy, with many degrees between PSD and ESD.

The CEM paradigm was a visual success and CEM drawings made in Excel became a de-facto standard. Some vendors followed with specialised CEM programming tools to generate control code for their respective platforms. See Figure 1 for an example of the CEM format.



Application Unit Shutdown Matrix EXAMPLE logic

							NO	21	21	21	21	21		21	21				21	21			21	21	
							NW	11	11	11	11	11		11	11				11	11			11	11	
							TNo							0	0				11	10			11	10	
							Al+							L	L				L	L			L	L	
							Alarm	Level	Level	Level	Level	Level		NE	NE				NE	NE			ND	ND	
							Type	ESD	ESD	ESD	ESD	ESD		DO	DO				DO	DO			DO	DO	
							Name	XX-ESD1	XX-ESD2	XX-ESD3	XX-PSD4.1	XX-PSD4.2		XX-OUTPUT01	XX-OUTPUT02				XX-VALVE01	XX-VALVE01-INT			XX-VALVE01	XX-VALVE01-INT	
							No	1	2	3	4	5		1	2				6	7			10	11	
No	Name	Type	Al+	NW	NO	Level																			
1	XX-ESD1	ESD		11	21				●					●											
2	XX-ESD2	ESD		11	21					●					●										
3	XX-ESD3	ESD		11	21						●	●													
4	XX-PSD4.1	ESD		11	21														●	●					
5	XX-PSD4.2	ESD		11	21																		●	●	
1	XX-ACT-ESD1	Alarm	IL	11	21	XX-ESD1		●						●	●				●	●			●	●	
3	XX-ACT-ESD2	Alarm	IL	11	21	XX-ESD2			●						●				●	●			●	●	
6	XX-INPUT01	Alarm	IL	11	21									●											
7	XX-INPUT02	Alarm	IL	11	21										●										
10	XX-ACT-VALVE01	Alarm	IL	11	21														●	●					
12	XX-ACT-VALVE02	Alarm	IL	11	21																		●	●	
14	XX-RESET-OUTP			11	21									■	■										

Fig. 1. Sample from a typical early CE Matrix

1.1 Trends in ESD/PSD Systems

The CEM format became a popular representation but without the support of a formal standard a variety of formats proliferated, even between different lifecycle stages of the same project. Recreating the same CEM in different formats introduced delays and increased the potential for misunderstandings and errors.

The Cause and Effect Matrix paradigm had become a victim of its own success.

In addition to the proliferation of data formats and visual formats, the CEM paradigm has also been under growing pressure to cope with these important trends in industry.

- **Size and complexity:** ESD/PSD systems comprise a large and ever-increasing number of tags, arranged in an increasingly complex hierarchy of cascading shutdown levels, with special cascade inhibit logic.
- **Safety compliance:** ESD/PSD programmable systems have high safety integrity (SIL) and the accompanying compliance burden is increasing as

regulatory demands become stricter Ref [IEC-61511]. In the operational phase, periodic proof-testing reports of the ESD/PSD are also mandatory.

- **Common Control Platforms:** The trend toward using commercial off-the shelf (COTS) hardware platforms and use of common control software libraries even for ESD/PSD systems with high safety integrity (SIL) requirements.
- **Decision support:** The need for operators to quickly trace backward to determine the cause(s) of a trip in an ESD/PSD system.
- **Additional signal processing:** ESD/PSD programmers want the freedom re-configure the input cause signals, add additional logic before setting the outputs.
- **Spanning process areas:** ESD/PSD systems spans many more diverse process areas, engineered by different contractors and teams.

2 Single Source Engineering

In their work with engineers responsible for large shutdown implementations, the authors discovered the benefits of "single-source engineering" (SSE) as a way of mitigating the problems created by the trends listed above. They offer the following recommendations, whose application to those problems are discussed in the following paragraphs of this paper.

- Single representation for CEM applications & visual displays: agree on a single standard visual "language" for representing CE matrices, from which the operator display and the 1131 code is co-generated.
- Single source of data for of CEM lifecycle activities: use a portable document as a single source of data for all activities throughout the lifecycle.

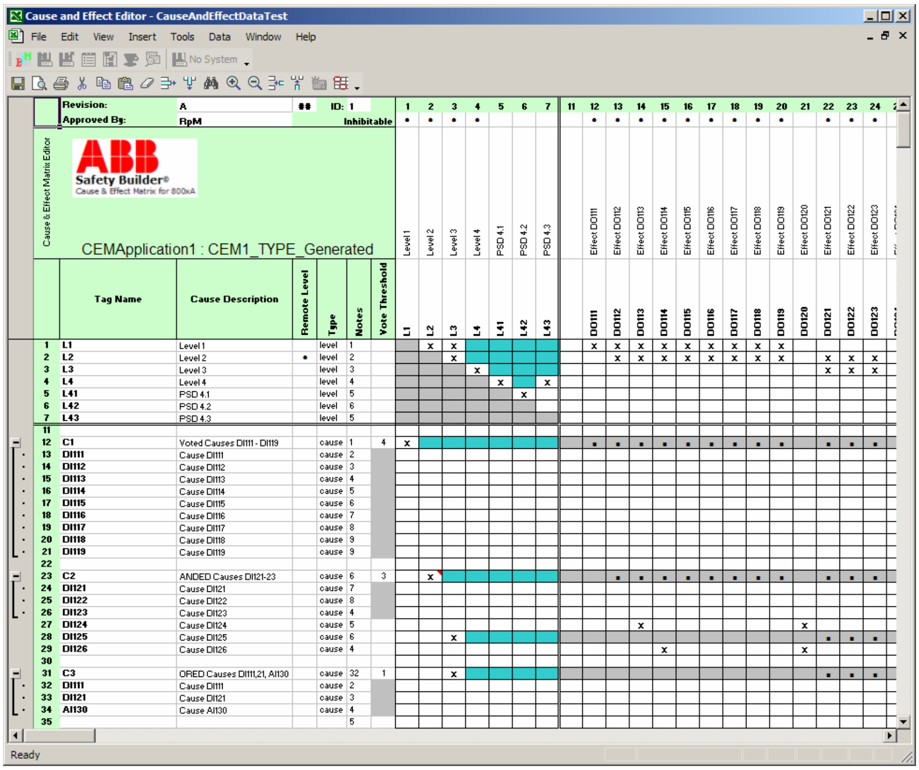
2.1 Size and Complexity

ESD/PSD systems comprise a large and ever-increasing number of tags, arranged in an increasingly complex hierarchy of cascading shutdown levels, with special cascade inhibit logic.Ref [DNV1]

The size and complexity of ESD/PSD systems have dramatically increased the effort of coding and maintaining such an application directly in one of the standard 1131 languages. A generative approach suggests itself, where 1131 code objects are automatically created based on the visual layout in the CEM diagram. The authors approve of vendors which offer such tools; programming at a higher level of abstraction is an effective way to reduce implementation errors and ease application maintenance.

The authors tool development layout is shown in the figure below. This matrix view uses text, colour and shading patterns to show direct and indirect trips as the result of cascading level logic. These indirect effects would be difficult to trace if the logic were programmed directly in control code structured text, for example.

Generative techniques should not stop at the control code; re-creating the logic for an operator display is also a very time-consuming and error-prone task. The author's


Cause and Effect Editor - CauseAndEffectDataTest
File
Edit
View
Insert
Tools
Data
Window
Help
No System
Safety Builder®
Cause & Effect Matrix for 800xA
Cause & Effect Matrix Editor
Ready

Revision: A ## ID: 1
Approved By: RpM

CEMApplication1 : CEM1_TYPE_Generated

	Tag Name	Cause Description	Remote Level	Type	Notes	Vote Threshold	1	2	3	4	5	6	7	11	12	13	14	15	16	17	18	19	20	21	22	23	24
						Inhibitable	•	•	•	•					•	•	•	•	•	•	•	•	•		•	•	•
							Level 1	Level 2	Level 3	Level 4	PSD 4.1	PSD 4.2	PSD 4.3		Effect DO111	Effect DO112	Effect DO113	Effect DO114	Effect DO115	Effect DO116	Effect DO117	Effect DO118	Effect DO119	Effect DO120	Effect DO121	Effect DO122	Effect DO123
							L1	L2	L3	L4	L41	L42	L43		DO111	DO112	DO113	DO114	DO115	DO116	DO117	DO118	DO119	DO120	DO121	DO122	DO123
1	L1	Level 1		level	1			x	x	🟦	🟦	🟦	🟦		x	x	x	x	x	x	x	x	x				
2	L2	Level 2	•	level	2				x	🟦	🟦	🟦	🟦			x	x	x	x	x	x	x	x		x	x	x
3	L3	Level 3		level	3					x	🟦	🟦	🟦												x	x	x
4	L4	Level 4		level	4						x	🟦	x														
5	L41	PSD 4.1		level	5							x															
6	L42	PSD 4.2		level	6																						
7	L43	PSD 4.3		level	5																						
11																											
12	C1	Voted Causes DI111 - DI119		cause	1	4	x	🟦	🟦	🟦	🟦	🟦	🟦		▪	▪	▪	▪	▪	▪	▪	▪	▪		▪	▪	▪
13	DI111	Cause DI111		cause	2																						
14	DI112	Cause DI112		cause	3																						
15	DI113	Cause DI113		cause	4																						
16	DI114	Cause DI114		cause	5																						
17	DI115	Cause DI115		cause	6																						
18	DI116	Cause DI116		cause	7																						
19	DI117	Cause DI117		cause	8																						
20	DI118	Cause DI118		cause	9																						
21	DI119	Cause DI119		cause	9																						
22																											
23	C2	ANDED Causes DI121-23		cause	6	3		x	🟦	🟦	🟦	🟦	🟦			▪	▪	▪	▪	▪	▪	▪	▪		▪	▪	▪
24	DI121	Cause DI121		cause	7																						
25	DI122	Cause DI122		cause	8																						
26	DI123	Cause DI123		cause	4																						
27	DI124	Cause DI124		cause	5												x							x			
28	DI125	Cause DI125		cause	6				x	🟦	🟦	🟦	🟦												▪	▪	▪
29	DI126	Cause DI126		cause	4													x						x			
30																											
31	C3	ORED Causes DI111,21, AI130		cause	32	1			x	🟦	🟦	🟦	🟦												▪	▪	▪
32	DI111	Cause DI111		cause	2																						
33	DI121	Cause DI121		cause	3																						
34	AI130	Cause AI130		cause	4																						
35					5																						

Fig. 2. A CEM editor using colour to show cascade logic

approach is to co-generate the display by linking display artifacts with the generated code variables to provide an online view of current status.

2.2 Safety Compliance

ESD/PSD programmable systems have high safety integrity (SIL) and the accompanying compliance burden is increasing as regulatory demands become stricter Ref [IEC-61511]. In the operational phase, periodic proof-testing reports of the ESD/PSD are also mandatory.

When the CEM drawing of the ESD/PSD system has become the subject of formal integrity approval by national authorities, then there is strong motivation to preserve its format throughout the engineering lifecycle. In this way, suppliers, operators and engineers have a common understanding of the CEM shutdown behaviour, based on a common visual representation. This approach assists "cognitive recognition" Ref [HCI2].

In the operational phase, periodic proof-testing reports of the ESD/PSD are also mandatory. Such reports are difficult to configure and maintain without access to the original CEM data structures from the design phase.

The solution to both of the above problems is to make the approved CEM drawing the single source of engineering data for all engineering and operational activities

When the document is approved then it can be argued that all subsequent transformations based on that document also inherit this approval. In the authors tool development, a single document serves as the basis for all lifecycle activities, including the operator display. This document can then be input to an editor/compiler capable of generating IEC-61131 compliant control code objects Ref [IEC-61131] and the operator display.

The advantages of having a "single source of truth" are many (Refs [CE1] , [CE2]), and these assume an even greater importance for the development of safety-critical systems. The authors suggest that this principle should be encouraged in future drafts of both corporate and international safety standards.

Using a single document throughout the engineering process and then taking this same document into the daily operational phase of a system will reduce synchronisation problems. Version checking and inconsistency reporting between the original documentation and the online operator display: By generating the control logic from the master document, a version number can be introduced to the control logic and later used online to check that the operator display has the same version number as the code running in the controller. This will help to ensure that the operator display is always up to date with the control logic: the display matches the plant "as built".

A single source approach enables printing of online status in a format that is identical to the design documentation: This aids the task of tracing trip signals and documenting shutdown situations in a consistent way. It is also very useful to be able to include information like versions, build numbers, approvals and engineering and operator notes in the online display.

2.3 Common Control Platforms

There is an industry-wide trend trend toward using commercial off-the shelf (COTS) hardware platforms and the use of common control software libraries even for ESD/PSD systems with high safety integrity (SIL) requirements. This trend will soon extend to safety fieldbusses [CE1].

This trend means that modern safety solutions must conform to standard languages, interoperate with standard device control software libraries and standard operator displays. Most importantly, it means that ESD/PSD safety systems can no longer assume exclusive ownership of the devices and must accommodate a wider and "wilder" range of configurations. The downside of this trend has the potential to make the engineering and operation of ESD/PSD systems more difficult and potentially increase the chances of design errors.

The benefits of using a single platform for safety and non-safety applications can be realised if the critical parts of that platform are safety certified. For control software this means selecting a relevant subset of 1131 function blocks or control-modules and ensuring that these are SIL-compliant. This subset should be sufficient to cover the needs of ESD/PSD programming. These safety status of these types should be clearly visible within the programming tool and that tool's compiler should check that their usage is consistent with the safety level of the entire application and the target controller. This approach allows safety engineering to flow together with the basic process control system engineering.

Implicit in this recommendation is the use of pre-defined 1131 types or classes, stored in tested and approved libraries. This type-based approach is especially important for safety programming, whether it is manual or generative, since it builds on approved and version-controlled code.

The benefits of standardisation can be extended by encapsulating all the variety of device types (transmitters, valves, motors etc.) inside control object types which expose their internal states in a standard way, for example in a bit pattern of a defined 32-bit variable. Ref [PCCUG]

2.4 Decision Support

As the complexity of ESD/PSD systems has increased, so has the need to assist operators in quickly tracing backward to determine the cause(s) of a trip, without having to deduce the cause from many different documents and inputs Ref: [HCI]. Operators find it increasingly difficult to understand a trip situation and trace backward to the originating causes in real-time.

The solution to this problem is to provide the operators with online displays which allow him/her to see the online status of the ESD/PSD system at any level of aggregation, with the freedom to drill down to individual process areas or devices. At the lowest device level, the operator is presented with an online "faceplate" display showing the device signal pathway traced back to one or more input devices.

To achieve this desirable functionality requires that all the signal pathways from output backward to input must be charted and embedded in the operator display logic. Typically this signal flow information is hidden within the temporary data structures of the 1131 code compiler and is not available to any other engineering tools.

The authors found that the most robust way to re-create this path information was to adopt the approach pioneered by a team led by Alan Munns of ABB in the UK, called the Priority Command Concept (PCC). The PCC is an open concept for 1131 type design. To be PCC-compliant, a control type (or "class") must expose its instance name and its internal state in data elements NAME and ACTION, respectively. A further requirement is that each PCC type must propagate the name of the PCC instances to which it is connected, both up- and down-stream. This propagation is done via 1131 string operations performed once in the first scan after start-up in the controller.

Each PCC instance knows its neighbors: this allows operator viewer tools to trace the signal pathway from output to input device, regardless of the number of intermediate control objects. These tools can automatically create a dynamic visual display element which is inserted into the view of the output device (See Figure below).

The operator can access any of the devices participating in the CEM matrix by right-clicking on the display itself and choosing the "faceplate" item as shown in the following figure.

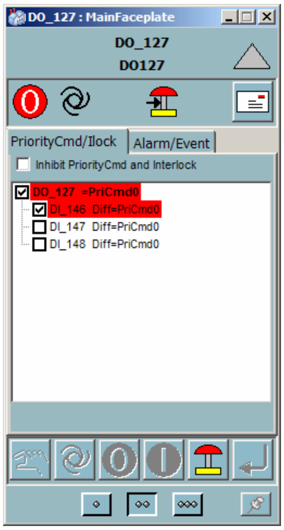


Fig. 3. Operator output signal faceplate showing signal pathway from input

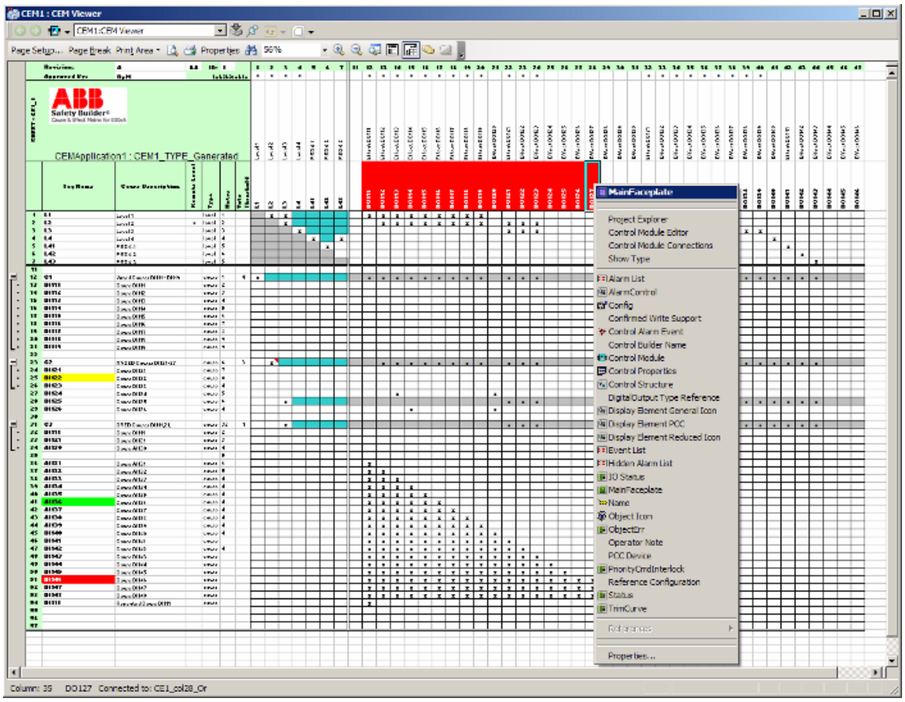


Fig. 4. CEM display for operator showing quick access to signal object

At the highest level, the user can navigate between the various process areas represented as different parts of the CEM matrix. The navigation display (shown in the figure below) shows these areas to the operator and are color-coded with the current status of all their contained device signals. The operator can click on the area "button" and be presented with the CEM matrix for that area, as shown in the previous figure.

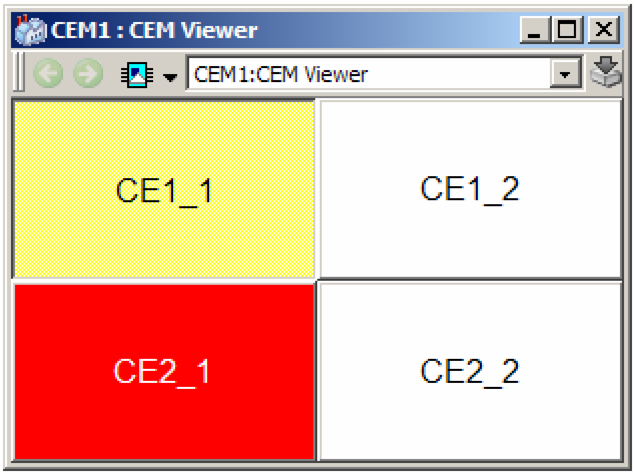


Fig. 5. Operator's top-level CEM Navigation display

The relationships between the various displays are shown in the following Figure.

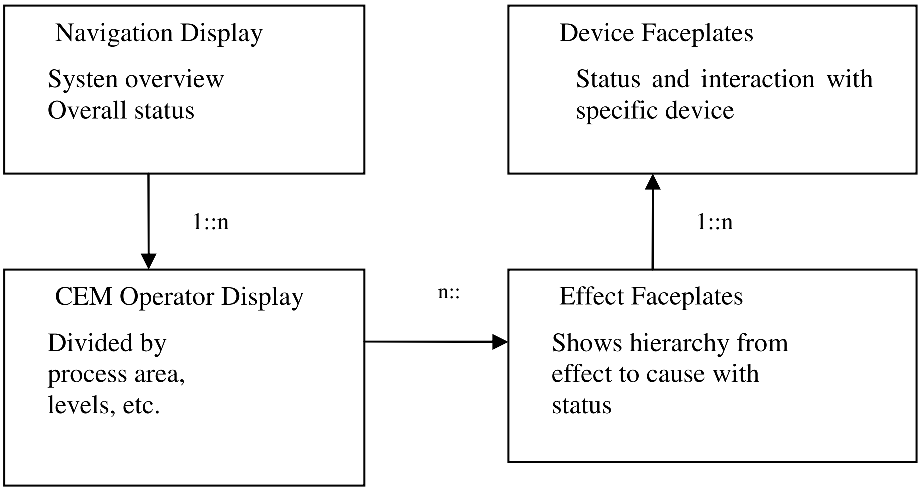

Navigation Display
Systen overview
Overall status
Device Faceplates
Status and interaction with specific device
1::n
1::n
CEM Operator Display
Divided by process area, levels, etc.
n::
Effect Faceplates
Shows hierarchy from effect to cause with status

Fig. 6. Relationships between operator CEM displays

2.5 Additional Signal Processing

ESD/PSD programmers want the freedom re-configure the input cause signals, add additional logic before setting the outputs. The previously clean and simple CEM grid format sprouted many footnotes and comments as designers struggled to capture special logic cases on the crowded matrix grid. An additional problem is the quality of the generated code, which in most cases is custom IEC-61511 structured text.

The solution to this problem is three-fold: firstly, vendors and industry should agree on a set of functionality which satisfies the needs of most CEM engineers. In particular, voting arrangements should be a standard part of such a proposed CEM language. The authors propose that the CEM language syntax be limited to these basic functions:

- Signal Name & Description
- Trip ("X")
- Inversion
- Normal & Cascade Inhibit
- Reset
- Time Delay
- Voting (NooM)
- Comments

Secondly, analog thresholding, boolean latching and other related configuration work should be encapsulated in the signal device object, so that the CEM is freed to do what it does best: show the routing cleanly and intuitively.

Custom textual coding nullifies one of the main benefits of single source engineering. So the final recommendation here is to use a control library in which even simple "routing" operations such as AND / OR / SPLIT are SIL- marked object types. These can be configured safely by the CEM editor user and then instantiated by the CEM editor's generation function, thus ensuring higher quality executable code. In the approach taken by the authors, these control object types simply route the originating device data, packaged within a standardised bit pattern of a 32-bit variable data element. This data type contains a data word (ACTION) with a standard bit pattern for commanded action, inhibit, connected and a few other states. The simple routing object type instances operate on this data.

2.6 Spanning Process Areas

ESD/PSD systems span many more diverse process areas, engineered by different contractors and teams.

Single source engineering is challenging in a multi-disciplinary and geographically dispersed engineering environment. Vendors who claim to support this principle offer centralised database or object stores, but access to these are complicated by the need for special client software and the ability to cross corporate firewalls.

The authors believe that the internet points the way to a more flexible architecture based on exchange of a document in a standard format which contains its own validation logic. This validation could be either in the form of an XML schema or via a component (such as ActiveX) which is embedded in the document. Such a document is highly portable: it can be exchanged via email between all the partners in a large project, without difficulties caused by firewalls, licensing, and installation and setting up special access permissions.

A typical use case illustrating this solution is when a contractor can specify CEM connections in a portable document without being forces to install any special software by the control system supplier. He/She then sends the document via email to the control system supplier who can then automatically generate the control code and the operator display without modifying or translating the original CEM document.

Finally, automation vendors are urged to now agree on a common visual and textual representation which should be submitted to a standards-setting body for approval. Taken together, these are syntax rules for a proposed common CEM "language". The textual representation should be in the form of an XML schema. The accompanying visual standard should reflect the contents of the XML representation in a universally recognisable way, which is intelligible to CEM programmers and operators.

3 Conclusion

Let us conclude by marking up our original list of problems with the recommendations:

- **Size and Complexity:** ESD/PSD systems comprise a large and ever-increasing number of tags, arranged in an increasingly complex hierarchy of cascading shutdown levels, with special cascade inhibit logic.
    - Program directly in the CEM matrix and use tools to generate both the control code and the operator graphic displays.
- **Safety Compliance:** ESD/PSD programmable systems have high safety integrity (SIL) and the accompanying compliance burden is increasing as regulatory demands become stricter Ref [IEC-61511]. In the operational phase, periodic proof-testing reports of the ESD/PSD are also mandatory.
    - Use a single CEM data source for lifecycle compliance activities.
- **Common Control Platforms:** The trend toward using commercial off-the sheflf (COTS) hardware platforms and use of common control software libraries even for ESD/PSD systems with high safety integrity (SIL) requirements.
    - Use a single common library with a subset of SIL-approved types which share a common interface with other types.
- **Decision Support:** The need for operators to quickly trace backward to determine the cause(s) of a trip in an ESD/PSD system.
    - Use object types which support upstream signal tracing and tools which can generate displays showing the active device signal pathways.
- **Additional Signal Processing:** ESD/PSD programmers want the freedom re-configure the input cause signals, add additional logic before setting the outputs.
    - Use an approach which allows reconfiguration and additional routing logic based on types, not loose code.

- **Spanning Process Areas:** ESD/PSD systems spans many more diverse process areas, engineered by different contractors and teams.
    - Use a portable document which can be easily shared amongst project teams.
    - Use a single visual representation "language"; make it a standard for programming by providing tools to generate the required 1131 code objects.

Acknowledgements

- Alan Munns, ABB UK, leader of the PCDeviceLib (PCC) development team, for his permission to cover the benefits of PCDeviceLib concepts in this paper.
- Steinar Engh , ABB Norway, for his valuable feedback as an experienced cause and effect matrix designer and user.

References

[IEC-61511]	International Standard IEC-61511 Functional safety –Safety instrumented systems for the process industry sector, International Electrotechnical Commission Geneva , Switzerland 2003
[IEC-61131]	International Standard IEC-61131 Programmable controllers, International Electrotechnical Commission Geneva , Switzerland 2003
[PCCUG]	PCDeviceLib User Documentation , ABB document # 3BGB001947D0063, ABB 2004 .
[CE1]	Control Engineering Magazine "Safety Networks" , 1/12/2004
[DNV1]	Offshore Standard DNV-OS-A101, "Safety Principles and Arrangements", 2001, Det Norske Veritas (DNV).
[HCI1]	Carroll, JM ; Human Computer Interaction in the New Millenium", Addison Wesley, New York, 2001
[HCI2]	Preece, J ; Human Computer Interaction", Addison Wesley, New York, 1999
[CE1]	Control Engineering , Sept. 2003, " Maintaining a Single Source of Truth."
[CE2]	ARC Insight Oct 2004, "Leverage Engineering & Design Information to Improve Plant Performance"

Combining Extended UML Models and Formal Methods to Analyze Real-Time Systems

Nawal Addouche[1], Christian Antoine[2], and Jacky Montmain[2]

[1] Ecole des mines d'Alès, Parc Scientifique Georges Besse,
30035 Nîmes, France
nawal.addouche@ema.fr
[2] URC CEA-EMA, Parc Scientifique Georges Besse,
30035 Nîmes, France
{christian.antoine, jacky.montmain}@ema.fr

Abstract. In the paper, we present a methodology developed in order to verify probabilistic temporal properties related to dependability of real-time systems. The methodology is made of three essential steps. The first one is a UML profile called DAMRTS (Dependability Analysis Models for Real-Time Systems) designed using GME tool. The aim is to model a real-time system with qualitative and quantitative information related to its quality of service. In this profile, UML statecharts are used to represent the system behavior. An extension is introduced with probabilities, real-time requirements and nondeterministic choices. The second one proposes a translation from the extended UML statecharts to probabilistic timed automata (PTAs). In this step, global clocks are used to represent synchronization of concurrent UML statecharts in probabilistic timed automata. The last one concerns a probabilistic model checking with PRISM tool. This requires specification of dependability properties with a suitable temporal logic.

1 Introduction

Several approaches have already been explored to introduce quantitative information in the dynamic UML models. A stochastic extension of UML statechart diagrams is proposed in [7]. It is based on a set of stochastic clocks which can be used as guards for transitions. The clock value is given by a random variable with specified distribution function. Other approaches are also proposed to formalize UML models which are extended with quantitative information. Dynamic UML models are formalised with stochastic Petri nets in [15], with stochastic process algebra in [5] or with continuous time Markov Chains such as we proposed it in [2]. This one is adequate for the performance evaluation and the verification of some dependability properties. However, the formal model contains only rates. Then, it is not suitable for modeling real-time systems. Different models exist to describe real-time systems such as timed automata [3] which have a clear semantics and for which a tool support for automatic verification (Uppaal, Kronos) is available.

The Unified Modeling Language (UML) [16] which becomes an official standard of the Object Management Group (OMG) is widely adopted in industry. This semiformal language, easy-understood and well-established design notation in the software engineering community, is extended to support the aspect-oriented design for a system. UML support many application domains and provides a common notation independent of the kind of systems that are developed.

To combine the advantages of intuitive modeling by UML with formal verification, we chose the approach which consists to transform UML models into the input language of an existing model checker. Input of a model checker is a formal description of a system. For formal analysis, it is necessary to define what kind of semantic requirements are implied by the domain and what kind of semantics that easily allows translation into the formal model to be analyzed by the model checker. Our contribution includes:

- Definition of the profile DAMRTS [1] for modeling and analyzing real-time systems: a class diagram is proposed to represent static model with quality of service of system components; the conventional UML statecharts are extended with probabilities and real-time requirements,
- Specification of the nondeterminism in extended statecharts and synchronization of concurrent statecharts,
- Métamodeling with the Generic modeling Environment (GME) to construct the proposed profile,
- Translation of extended UML statecharts to probabilistic timed automata: global clocks are defined to represent synchronization of UML statecharts.

In section 2, the methodology of real-time systems analysis is described. The dynamic view of UML models is presented in sections 3 and 4. We present in section 5, real-time constraints of an assembly chain as well as their behavioral UML models as defined in the profile DAMRTS. This one is designed using GME tool as presented in section 6. The behavioral UML models are nondeterministic, with probabilistic transitions and real-time aspects. That make possible to translate them into probabilistic timed automata as given in section 7. The translation process of resulting models is also described in this section. In section 8, we give an overview on the type of properties that can be checked. We conclude with section 9.

2 General Methodology

In order to have UML accepted by the real-time development community, the OMG group has proposed a profile called “Schedulability, Performance and Time” [17] for real-time systems. In this profile, some supports are introduced in UML to capture a maximum of real-time requirements and to perform the real-time development tasks directly on UML models. Beside the usual analysis and design stages, scheduling analysis, performance evaluation and formal verification of critical properties are included. However, the two last activities are partially covered because “quality of service” requirements are introduced without a clear indication about the formal verification of this type of properties. Adapted tools to formal verification or performance evaluation on these UML models are not yet available.

For the reasons indicated above, a new profile called DAMRTS is proposed to analyze and verify dependability properties of real-time systems [1]. It represents an extension to the reference metamodels of the OMG profile “Schedulability, Performance and Time” [17]. The first aim is to be compliant with the standard OMG profile. The second one is to provide concepts that enable to specify a real-time system with its real-time constraints and probabilistic information. A behavioral UML models are proposed with a formal semantics served to probabilistic model checking [10]. It is developed with probabilities and real-time aspects, resulting in probabilistic timed automata as semantics models. These models are used to verify probabilistic temporal properties related to the dependability of real-time systems.

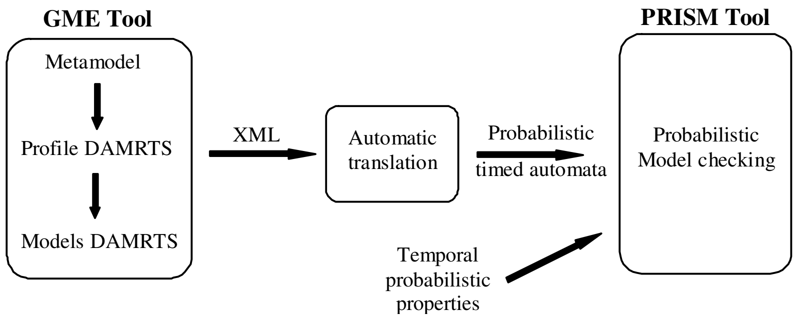

GME Tool
Metamodel
Profile DAMRTS
Models DAMRTS
XML
Automatic translation
Probabilistic timed automata
Temporal probabilistic properties
PRISM Tool
Probabilistic Model checking

Fig. 1. Global methodology

As depicted in fig.1, the proposed approach is presented with the essential steps that allow associating a formal method to UML models extended with probabilities and time. The GME tool is used to construct metamodels specifying the modeling paradigm (modeling language) of our application domain. The modeling paradigm contains all syntactic, semantic and presentation information regarding the domain of real-time systems dependability. This is developed in section 6.

Once the profile DAMRTS is built, we model the real-time system. The output of GME tool is a file having an extended XML format. The DAMRTS models are then exported in XML format for which an automatic translation is applied to transform UML behavioral models into probabilistic timed automata as it is detailed in section 7.3. The PRISM tool is then used to verify the properties of real-time system.

3 Modeling with Extended UML

To represent dynamic aspects of the system, extended UML statecharts and collaboration diagrams are used. Combination of these two diagrams allows representing all system interactions. Indeed, the collaboration model describes external interactions between objects whereas UML statecharts diagrams represent how an instance of a class reacts to an event occurrence.

A collaboration diagram consists of objects and associations that describe how the objects communicate. It represents the structural organisation of objects which exchanges messages. In the DAMRTS profile, the signals and the orders are the two types of messages taken into account. The first ones are sent from objects of Sensor classes to objects of Controller class. The second ones are sent from instances of Controller class to instances of Effector classes (see fig 3).

The proposed statecharts allow expressing events with probabilities and actions with real-time constraints. The nondeterminism and synchronization are defined as follows:

Nondeterminism. Nondeterministic choices can be specified in transition systems by having several transitions leaving from the same state. It is used when we wish to incorporate several potential system behaviors in a model. Nondeterminism is used for several purposes. As it is specified in [6] and [18], it is used to represent phenomena such as:

Unknown scheduling in concurrent systems. When a system consists of several components running in parallel, we often do not make any assumptions on the relative speeds of the components, because we want the application to work no matter what these relative speeds are. Therefore nondeterminism is essential to define the parallel composition operator, where we model the choice of which system take the next step as a nondeterministic choice.

External environment. A system interacts with its environment via its external actions. When modeling a system, we can not predicate how the environment will behave (failures, abnormal functioning). Therefore the possible interactions with the environment are modeled by nondeterministic choices.

Uncertainty in probabilities and the expected times. Sometimes it is not possible to obtain exact information about the system to be modeled. When the exact duration of an action or the exact probability of an event is not known exactly but only with a lower and upper bound. In this case, all possible values are incorporated by nondeterministic choices.

Synchronisation. The extended UML statecharts are allowed to communicate with each other in well-defined manners. The communication and synchronization method are presented as follow:

- One UML statechart may create an event as a result of a transition that is consumed by another UML statechart.
- A guard may be used to test if another UML statechart is in a certain state before allowing a transition to occur to the guarded state.

4 Extended UML Statecharts

In UML, each class has an optional statechart which describes the behavior of its instances (the objects). This statechart receives events from other statecharts and reacts to them. The reactions include sending of the new events to other objects and executing of internal methods on the object. The communications between components of the system are modeled as events. Exchanged signals and orders as well as random events (e.g. undesired and lost signals) are represented as events associated to a discrete probability distribution.

The syntax of UML statecharts, defined in the standard UML [16] is extended as presented below. The operational semantics of UML statechart is inspired from [9] and extended with real-time and probabilistic aspects as presented in [1]. The informal interpretation of extended UML statecharts is based on a set of nodes and a set of edges. An edge is presented by the following syntax:

Edge: = Event [Guard] / Action
Event: = Event name (Probability)
Guard: = Boolean Expression
Action: = Operation name (Arguments) [Duration, deadline]

Event represents received signals, sent orders or random events with their associated probability. Guard is a boolean expression which represents either *AND-composition* or *OR-composition* states related to degraded or failure states of other objects. The compositions are excerpted from a faults tree analysis of the system [1]. Action expresses operation execution or sending messages to other objects. They are not instantaneous but have duration or deadline. Transitions between states are probabilistic. When two transitions are enabled, the choice is nondeterministic.

5 Example of an Assembly Chain

This example presents an automated chain assembly of electrical micro-motors. It is excerpted from a European project named PABADIS (Plant Automation BAsed on DIstributed Systems) [14]. This one deals with a flexible and a reconfigurable system designed for production of different types of micro-motors. Fig 2 represents the controlled system.

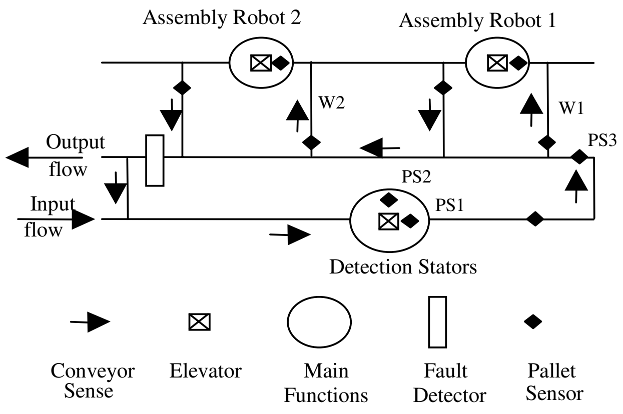


Fig. 2. Assembly chain of micro-motors

Micro-motors consist to stators and rotors. The first are transported to assembly robots, on pallets via a conveyor system and seconds are available into stocks near each robot. A set of pallets containing stators moves along the conveyor belt. These are detected by pallet sensors PSi at different levels of the conveyor system. When assembly of micro-motors is completed, the pallets then move into a fault detection station where a camera detects the possible assembly faults. Set of PLC (Programmable Logic Controller) and PC composes the control system.

The assembly robots work in parallel. Let us consider a stators pallet arrives at the level of assembly robots and detected by PS3. If the two robots are both idle, the pallet is arbitrary send to one of the waiting areas w1 or w2 showed in fig 2. If the robots are both busy, the controller send information request to robots. These send the information about the assembly state. The pallet is then leaded to the robot which will be the idle first. This behavior is modeled as given in the statecharts of fig 4 and fig 5.

5.1 Collaboration Diagram

In the collaboration diagram of fig 3, interactions between objects are presented. Exchanged messages describe the signals (S.PPi) and (S.Fault), respectively sent from pallet sensors (*PSi* for *sensor* i) and *Fault detector* objects to the *Controller*. They also represent orders sent from *Controller* to *Robot 1, Robot 2* and *Elevator* objects. Our example presents a distributed system such that several controllers (PLC) interact to control the system functioning. To simplify, we represent in the collaboration diagram one *controller* object.

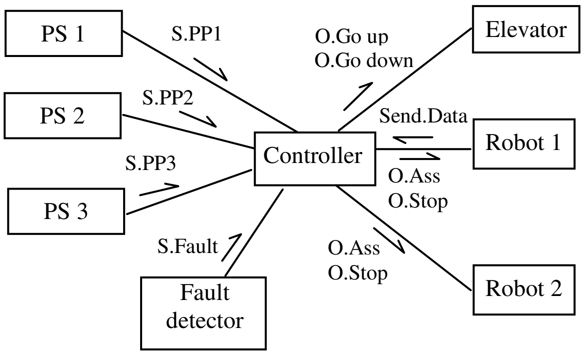


Fig. 3. Collaboration diagram

5.2 Extended UML Statecharts

Robot and *Controller* objects behavior are respectively modeled in fig 4 and fig 5. In Robot statechart (describes robot 1 or robot 2); assembly tasks as well as communication with controller are executed in parallel.

In substate A, the controller orders are modeled as events. In transition: “O.Ass (0.10) [PS1.Ds OR PS2.Ds]/ Assembly()[10s]”, guard expresses that the edge is enabled if one of sensors PS2 or PS3 is in degraded state Ds. The probability of sending an assembly order when one of sensors is in degraded state is evaluated to 0.10; the execution of the assembly operation, Ass () lasts 10s. Otherwise, the robot performs a correct assembly with probability 0.90. The guard “S.Active AND E.Active” represents the condition to leave the state Emergency stop: sensors and elevator must be in the state Active of their respective UML statecharts. When probability is not represented, it means it is equal to 1.

Nondeterminism is modeled at the level of the state, *Idle*. A probabilistic choice is used to represent the possibility of performing a correct or a faulty assembly. It is also possible that robot remains idle when there is absence of pallets (AP). Substate B, describes the controller requests and sending of data from robot to controller.

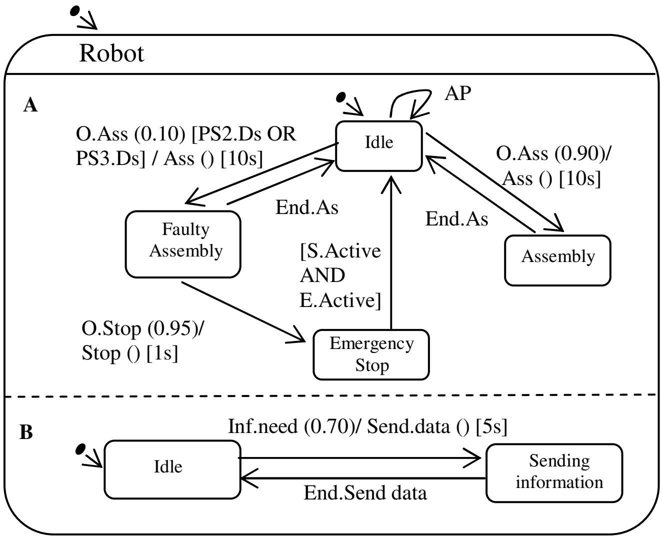

Robot
A
AP
Idle
O.Ass (0.10) [PS2.Ds OR PS3.Ds] / Ass () [10s]
O.Ass (0.90)/ Ass () [10s]
End.As
End.As
Faulty Assembly
Assembly
[S.Active AND E.Active]
O.Stop (0.95)/ Stop () [1s]
Emergency Stop
B
Inf.need (0.70)/ Send.data () [5s]
Idle
Sending information
End.Send data

Fig. 4. Robot statechart

Fig 5 describes controller behavior. When signal S.PP3 becomes true (sensor PS3 detects a pallet), the order Info.need is send to the robots. After receiving information, the controller sends the order of assembly for one of the two robots (which will be the idle first).

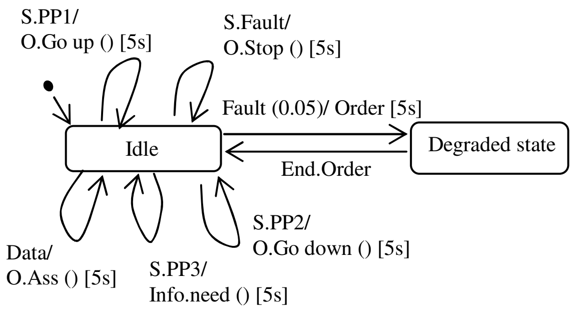


Fig. 5. Controller statechart

Among the malfunctions of controller, sending of undesired orders or lost orders are modeled in *Controller* statechart as random events, e.g. “Fault (0.05)/ Order [5s]”. The received sensor signals are presented as events and the sending of orders as actions with their associated deadlines. The edge “S.PP1/ O.Go up()[5s]”, expresses that when PS1 detects a pallet, order to go up from controller to elevator is send.

6 Metamodeling Using GME Tool

The Generic Modeling Environment (GME) developed at the institute for Software Integrated Systems at Vanderbilt University is a configurable toolkit for creating domain-specific modeling and program synthesis environments [13].

There is a metamodeling paradigm defined that configures GME for creating metamodels. These models are then automatically translated into GME configuration information through model interpretation. Once the metamodeling interpreter is operational, a meta-metamodel is created and the metamodeling paradigm is regenerated automatically [13]. The metamodeling paradigm is based on UML notation. The syntactic definitions are modeled using UML class diagrams and the static semantics are specified with constraints using the Object Constraint Language (OCL).

6.1 Modeling Concepts

The vocabulary of the domain-specific languages implemented by different GME configurations is based on a set of generic concepts built into GME itself. This one supports various concepts for building complex models.

Folders, FCOs (Models, Atoms, Sets, References, and Connections), Roles, Constraints and Aspects are the main concepts that are used to define a modeling paradigm. The First Class Objects (FCOs) used to represent entities and relations, form the core of the GME concepts. These generic concepts are not generally used at the same time. However, the choice is rather an important design decision. The concepts used in our metamodel are: Aspects, Models, Atoms and connections. These latter and the other quoted concepts are defined in [8] and [13].

6.2 Overview on the Metamodels of DAMRTS

The DAMRTS profile is a specific profile designed for dependability analysis of a real-time system. It is based on concepts defined in the profile SPT [17] with new stereotypes. Those are added to the metamodel in order to introduce particular dependability information. The malfunctions considered as undesirable events and their possible causes are modelled with stereotypes. The QoS is represented as attributes when it is about actions of resource classes (e.g. duration of actions, response time for a call action, etc.). It is also represented as a tagged value when it is about general QoS, like reliability and maintainability of resources [1].

To build the profile DAMRTS, the metamodeling paradigm based on UML is used. Three UML metamodels are created to represent class diagram, collaboration diagram and extended UML statecharts of a real-time system. Such as presented in fig 6, the metamodel of the class diagram contains the concept Atoms: sensor, effector (all components in contact with raw material such robots, conveyor belt, etc.) and controller.

The FCOs *Resource* and *Dependability* are defined to be hidden in the class diagram. We use the concept Attributes (which does not necessarily represent attributes in the class diagram) to define the QoS related to dependability of *Resource* as well as the methods send signals, send orders and actions related to the Atoms sensor, controller and effector (see fig 6). The Attributes of GME tool can have a set of specifications such as the data-type [8]. Then we specify the defined Attributes with integer or double according to whether it is of real-time data (deadline and duration of methods) or of probabilities assigned to undesired events.

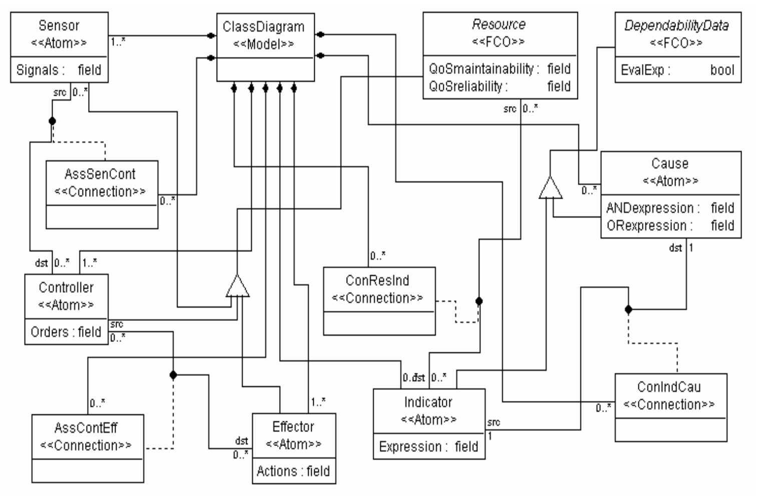


Fig. 6. Class diagram metamodel of profile DAMRTS

Each entity *Resource* is associated the Atom *Indicator* which represent the undesired events such failures. The Atom *Cause* has as attribute one or several logical expressions composed by elementary logical conditions linked by conjunctive and disjunctive connectors. This attribute is specified to be boolean. When one of the expressions is true, it indicates that associated failure became true.

7 Translating Extended UML Statecharts to PTAs

Timed automata are automata extended with clocks, positive real valued variables which increase uniformly with time, and whose nodes and edges are labeled with clocks constraints, respectively called invariants and guards. The invariant dictate when the automaton may remain in a node, letting time pass, and guards when the corresponding edge can be taken [3]. Probabilistic timed automata are a variant of timed automata extended with discrete probability distributions [11]. This type of automata has been chosen for formalizing extended UML statecharts because it takes into account dense time, nondeterminism and probabilistic choice as defined in the extended UML statecharts. They are also amenable to model check probabilistic temporal properties.

7.1 Principle of Translation

To translate extended UML statecharts to probabilistic timed automata, real-time constraints of actions are represented with clocks. Events are described with their probabilities on edges. The guards defined in proposed UML statecharts describe the active state of other objects.

In probabilistic timed automata, it is not really possible to observe the location of another component directly as the principle defined in our extended UML statecharts. However, a probabilistic timed automaton component A can check the location of another component B in the following way: component B is equipped with self-loop edges in all of its locations (or some of its locations), where the events of the self-loop edges would be different for each location. Therefore, in location L1, the probabilistic timed automaton B would have enabled a self-loop edge with an event which is unique to L1: “in-L1”, for example. Then component A, when it want to know whether B is in L1 or not, would try to synchronize on event “in-L1”. If synchronization is possible, then A knows that B is in L1 and can act accordingly; otherwise, it can do something else, knowing that B is not in L1.

7.2 Synchronization with Global Clocks

Synchronization between probabilistic timed automata components is done using edge-labeling events, as defined in [12]. One manner to synchronize probabilistic timed automata is to create a probabilistic timed automaton component which has a single clock which is never reset during the execution of the system. Then this clock could be regarded as a “global clock”. This component could then synchronize with the other components when the value of the global clock reaches certain values.

In fig 7, we give probabilistic timed automaton describing a sub-system of the assembly chain example: the robot behavior reacting to controller orders. The probabilities used in the example should in practice be obtained from statistical analysis of observed behavior.

In the probabilistic timed automaton, the sub-system consists to robot, controller and two clocks x and y. Atomic propositions, related to probabilistic timed automata of elevator and sensor, are included in nodes. “s:Ds, e:Active and s:Active” express guards of UML statecharts in fig 4. In initial state, both clocks x and y set to 0. The controller sends assembly order to robot in 5 time units. Then, the robot performs the assembly tasks with probability 0.90. After assembly takes 10 time units, the robot becomes idle. When one sensor is in degraded state, then robot performs a fault assembly with probability 0.10. When an order stop is send by the controller, the robot stop. It becomes idle when sensors and elevator are in active state.

7.3 Automatic Translation Process

Once the UML models are built with the profile DAMRTS, the different view of the real-time system are presented including dependability information. The following step consists to analyze the models. For lack of UML's models analysis tools, we chose a tool which allows analysis of models containing real-time and probabilistic information such as in the extended UML statecharts. The probabilistic model checker PRISM is then used [10].

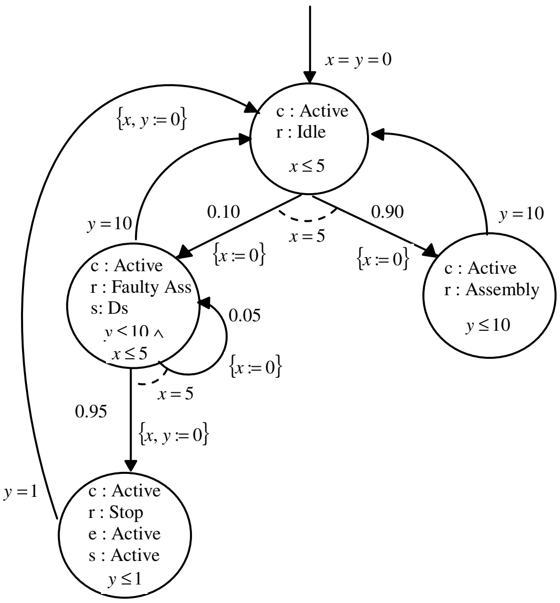


Fig. 7. Probabilistic timed automaton

To allow verification of probabilistic temporal properties, it is necessary to translate behavioral UML models from the GME tool to input model of PRISM tool. For this, an automatic translation is performed using the parser Xerces [19].

The statecharts UML models are exported to XML format. Syntactic analysis is applied on the XML files using the parser. Transformation rules are then defined to rewrite the XML nodes to PRISM language based on the Reactive Modules formalism [4]. This formal model is designed for concurrent systems and represents synchronous and asynchronous components in a uniform framework that supports compositional and hierarchical design and verification.

8 Probabilistic Model Checking

To verify dependability properties, the model checker PRISM is adopted [10]. This tool is designed for analysis of probabilistic models and supports various models such as Markov decision processes, discrete time Markov chains and continuous time Markov chains. The tool takes as input a description of a system written in PRISM language. It constructs the model from this description and computes the set of reachable states. It accepts specification in either the logic PCTL or CSL [11] depending on the model type. It then determines which states of the system satisfy each specification.

Some probabilistic properties related to our example are presented. Their informal specifications are given as follows:

Property 1: "The probability the robot carries out a faulty assembly is less than 1%".
Property 2: "In initial state, the probability that the robot remains in emergency stop until the elevator and the sensors are reactivated is at least 0.95".
Property 3: "Elevator remains in down position less than *k* units of time until the sensor 1 detects a pallet with a probability $\geq p$".

These dependability requirements are formally specified with PCTL. We use the variables: r, e and si, to respectively represent the states of the robot, elevator and sensors. The following are their PRISM specification language:

Property 1: P<0.1 [(r=2)]
Property 2: "init"$\Rightarrow$ P>0.95[(r=3) U (e=0) & (s1=0) & (s2=0) & (s3=0)]
Property 3: P=? [(e=0) $U^{\geq K}$ S.PP1= true].

The fundamental components of Prism language are *modules* and *variables*. A system is modeled with a number of modules which can interact with each other. A module contains a number of *local variables*. The values of these variables at any given time constitute state of the module. Global state of the system is determined by local states of all modules. Though the model checking method is automatic, it is confronted to the explosion of system states number. In our case, the difficulty of handling the models particularly depends of type of the *local variables* (integer, double, etc.) that we manipulate, combined with the number of these variables.

9 Conclusion

The approach used in our proposition is to enrich the UML model with the local quality of services parameters relevant to a specific analysis objective (for instance, failure/repair rates are associated with elements of UML model) and to automatically transform the relevant parts of the enriched UML models to probabilistic timed automata.

The advantage of the approach is that it is relatively easy to experienced UML users to create extended UML models and automatic translation made it possible to apply PRISM. The formal model is correct with respect to requirements of UML model. Writing properties with probabilistic temporal logic such as PCTL is not easy. In PRISM, syntax is proposed to express properties. This one is easier than that of probabilistic temporal logic.

Due to the denseness of time, the underlying semantic model of a probabilistic timed automaton is infinite, and hence effective decision procedure rely on building a finite quotient of the state space. In future works, the verification technique used, will be based on the generation of the forward reachability graph with Kronos, and model checking the obtained graph encoded as a Markov decision process with PRISM.

References

1. Addouche, N., Antoine, C., Montmain, J.: UML Models for Dependability Analysis of Real-Time Systems. In: Proc of SMC'04, The Hague, The Netherlands (2004)
2. Addouche, N., Antoine, C., Montmain, J.:"Formalisation of Quantitative UML models Using Continuous Time Markov Chains", Third Conference on Management and Control of Production and Logistics, Santiago, Chile (2004)
3. Alur, R., Dill, D.L.: A Theory of Timed Automata, Theoretical Computer Science, 126(2):183-235 (1994)
4. Alur, R., Henzinger, T.: Reactive Modules, In: Proc of LICS'96, IEEE Computer Society Press, New Jersey (1996), 207-218
5. Canevet, C., Gimore, S., Hillston, J., Stevens, P.: Performance Modelling with UML and Stochastic Process Algebra, In Proc of the Eighteenth Annual UK Performance Engineering Workshop (2002)
6. De Alfaro, L.: Formal Verification of Probabilistic Systems, PhD thesis, Standford University (1997)
7. Gnesi, S., Latella, D., Massink, M.: A Stochastic Extension of a Behavioural Subset of UML Statechart Diagrams, In: Proc of HASE'00, Albuquerque, New Mexico (2000)
8. ISIS.: GME 4 User's Manual, version 4.0, Institute for Software Integrated Systems, Vandertbilt University (2004), http://www.isis.vanderbilt.edu/Projects/gme/
9. Jansen, D.N., Hermanns, H., Katoen, J-P.: A Probabilistic Extension of UML Statecharts: Specification and Verification, FTRTFT 02, Oldenburg, Germany (2002) 355-374
10. Kwiatkowska, M., Norman, G., Parker, D.: Prism: Probabilistic Model Checker, In Proc.TOOLS 2002, volume 2324 of LNCS (2002), 200-204
11. Kwiatkowska, M.:Model Checking for Probability and Time: From Theory to Practice, In LICS 03, IEEE Computer Society Press (2003) 351-360
12. Kwiatkowska, M., Norman, G., Sproston, J.: Model Checking of Deadline Properties in the IEEE 1394 Fire Wire Root Contention Protocol, Formal Aspects of Computing (2003), 14(3), 295-318
13. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, Budapest, Hungary (2001)
14. Lüder, A., Peschke, J., Sauter, T., Deter, S., Diep, D.: Distributed Intelligence for Plant Automation on Multi-agent Systems: the PABADIS approach, Production Planning and Control (2004), 15(2), 201-212
15. Merseguer, J and J. Campos. (2002). A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In: Proc of WODES'02, pp. 295-302, IEEE Computer Society Press, Zaragoza, Spain.
16. OMG.: Unified Modeling Language Specification v.1.5, Formal / 03-03-01, (2003)
17. OMG.: UML Profile for Schedulability, Performance and Time v.1.1, Formal / 05-01-02, (2005)
18. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems, PhD thesis, Massachussetts Institute of Technology (1995)
19. XML.: The Apache XML Project. Xerces2 java parser 2.6.2 Release, The Apache Software Foundation (2004), http://xml.apache.org/xerces2-j/

Defining and Decomposing Safety Policy for Systems of Systems

Martin Hall-May and Tim Kelly

Department of Computer Science,
University of York, York, YO10 5DD, UK
{martin.hall-may, tim.kelly}@cs.york.ac.uk

Abstract. A 'system of systems' (SoS) comprises many other systems operating collectively with a shared purpose. Individual system autonomy can give rise to unpredictable, and potentially undesirable, emergent behaviour. A policy is a set of rules that bounds the behaviours of entities. Policy can be expressed at various levels of abstraction. By building on existing goal-based decomposition approaches this paper proposes policy as a means of achieving safety in SoS. The decomposition of policy to lower levels of abstraction must be carried out in a consistent, complete and systematic manner. The approach is agent-oriented and emphasises the recognition of contextual assumptions (such as knowledge of other agents' behaviour) in decomposing policy. To this end we present patterns of decomposition based on KAOS tactics of refinement. The application of these patterns, expressed in the Goal Structuring Notation, is illustrated using existing civil aerospace policy (the Rules of the Air Regulations).

1 Introduction

There exist systems whose constituent components are sufficiently complex and autonomous to be considered as systems in their own right and which operate collectively with a shared purpose. Many real *systems of systems* are geographically distributed and some of its component systems are mobile. Examples are numerous and include any permanent transport network (such as air, rail or road) as well as more short-lived SoS which may arise in network-centric warfare.

In such SoS the interactions between component systems are not constrained by physical design as in conventional monolithic systems. Since the SoS often comprise systems designed, manufactured and operated by various organisations, the set of possible interactions between any of the entities in the whole SoS cannot be known by any one individual. Such unpredictable interactions, if left unchecked, can lead to undesirable emergent behaviour, which may lead to accidents and loss of life. Some means is required to bound the behaviour of the system entities in such a way that no accidents occur. Defining a *safety policy* is the first step towards providing the necessary degree of control of interactions and coordination of responsibility.

Historically, interpretation of policy has relied upon human intelligence; hence loose and possibly ambiguous guidelines have been acceptable. This has not always been successful, as evidenced by the case in which two aircraft ostensibly operating according to policy were nevertheless involved in a fatal collision [1]. In this case it seems that the policy was not constrictive enough. In contrast, work-to-rule strikes expose flaws in overly conservative policies by reducing operational effectiveness.

An increasing desire to deploy unmanned and highly autonomous systems has brought to the fore the challenge of producing correct and complete safety policies. With systems such as unmanned air vehicles (UAVs) entering service we no longer have the luxury of relying on human flexibility and ingenuity to deal with vague or over-constraining policies. A way must be found to decompose high-level safety goals into policy 'statements' that are formulated in such a way that they can be implemented by man or machine.

This paper puts forward a number of patterns for decomposition. Section 2 provides an overview of policy. Section 3 presents the challenges of developing policy. Sections 4 and 5 describe the approach to supporting policy development. Section 6 outline the problems of defining a resilient policy. Finally there is some discussion of related and future work.

2 Overview of Policy

Policy describes the allowed envelope of an entity's actions, in that it defines behaviour that is both *permitted* and *required* from individual entities in order to be able to operate in a given environment (as described by the assumed context). To take a simple example as an illustration, consider a mother who asks her child to go to the corner shop to buy a pint of milk. She may lay down two rules with which the child must comply on this trip:

1. The child must not talk to strangers.
2. The child must use the pedestrian crossing when crossing the road.

The first of these rules defines what the child is allowed to do, specifically it proscribes conversation with people with whom the child is not previously acquainted. The second statement expresses the obligation that the child should take a safe route across the road, namely by using the pedestrian crossing.

Together these rules form a policy that guides the behaviour of the child on a journey to the corner shop. However the policy is orthogonal to the plan or mission of the child. It still holds regardless of whether the child is going to buy a loaf of bread or a dozen eggs, or not going to the corner shop at all.

Both rules are motivated by the desire that no harm should come to the child. Perhaps we have identified being in the path of an oncoming car and being in the company of untrustworthy (and hence potentially malevolent) individuals as hazards. However, even this simple policy is fraught with problems. Indeed, it demonstrates problems that face larger and technologically more complicated SoS. That is, the need to constrain and permit interactions.

3 Challenges of Developing Safety Policy for SoS

There are a number of challenges to developing a safety policy for a system of systems. The policy must take into account the following SoS characteristics:

- Wide variety of systems.
- Dynamic environment.
- Changing number of system entities.

The inability to address these issues fully leads to an assumed or implicit context in expressed policy statements. Indeed, the challenges for developing policy for SoS are in addition to the issues of formulating policy in a more general sense:

- Ambiguous nature of policy statements.
- No structured process of generating policy.
- Statements expressed at various levels of abstraction with no clear relationship between them.

All policy statements are expressed in the context of assumptions about the capabilities of the systems they address. Therefore a policy places a restriction on the type of systems that may form part of the SoS. For instance, the policy above requires that the child be able to recognise and operate (where necessary) a pedestrian crossing. For legacy systems this requirement may entail modifying the way they operate to comply with policy, together with the attendant costs this involves. In the case of human operated systems, it may involve retraining people to be aware of the new policy. If, however, a capability is required that is not already provided by an existing system, or new technology (e.g. UAVs) is replacing old, the policy indirectly places constraints on the design of these new systems.

Clearly, therefore, there is a relationship between safety policy and the requirements on individual system design as well as the configuration of the SoS. This relationship is not at first obvious and, what is more, the concepts are often conflated in current policy documents. Similarly, the formulation and adherence to safety policy can have a strong relationship with the safety arguments required within system safety cases. The details of this relationship is beyond the scope of this paper but is discussed further in [2].

4 Supporting Policy Development

The Goal Structuring Notation (GSN) [3] — typically used to construct safety cases — can be used to represent policy decomposition structures. In certain respects GSN is similar to another graphical notation, KAOS.

KAOS is a goal-oriented notation for representing requirements refinement hierarchies. However, KAOS adopts a more formal approach to the specification of goals in that it employs temporal logic to formulate expressions about requirements. Nevertheless, it is possible to adapt some of the methods from one technique to the other. Namely the work by Darimont and van Lamsweerde [4] detailing patterns of refinement.

Refinements in KAOS can be formally proven, whereas GSN (owing to the inductive nature of most safety arguments) does not attempt to formally prove decompositions. However, it has been suggested that argumentation techniques have a role to play in the engineering of emergent systems such as SoS [5]. Indeed, classical refinement is still a process of trial and error (*ibid.*). Using GSN allows documenting of the context under which the decomposition of the policy goals takes place. This means that any assumptions can be contested and that the decision processes are not hidden or implicit. They are therefore accountable and subject to scrutiny and change should they be found inadequate, or in the event of a change to the originally assumed context.

Policy cannot be formulated without consideration of the systems whose behaviour it is expected to influence. That is, there must exist a model of expectations about the agents and their environment. It is important to recognise and capture these expectations in terms of the context in which policy is expressed.

The policy model that is assumed in this work allows for a hierarchical structure of policy. The decomposition progresses from high-level, often state-based, goals down to action-oriented policy statements. GSN allows context to be captured at every level of the decomposition. It also explicitly documents the strategy by which the decomposition takes place. It is features such as these that KAOS lacks, and which we will make use of in the next section.

5 Patterns of Decomposition

In this section patterns are introduced that facilitate the process of decomposing policy from high-level safety goals down to implementable rules. These patterns are illustrated in GSN, however they have been inspired by work on tactics for requirements elaboration in KAOS. The reuse of common structures in GSN through the use of patterns has been recognised [6]. However, KAOS' patterns take advantage of the formal specification of requirements goals and can be formally derived and proven. Every pattern is proven once for all, hence every application of said pattern is correct.

Three tactics for developing patterns are identified by Darimont [7]:

– Agent-based decomposition
– Milestone-based decomposition
– Case-based decomposition

The patterns are described in more detail in the following sections and illustrated with examples from the civil aerospace Rules of the Air Regulations (RoA). The RoA [8] can be thought of as the policy that guides the behaviour of aircraft that wish to operate in the civil aerospace system of systems. It is expressed as a natural language document, which sets out a number of rules for the safe inter-operation of aircraft and air traffic control (ATC).

The RoA document sets out rules without the principles on which they were derived. The work shown in this paper represents an attempt to reverse engineer the RoA and, in so doing, rediscover the rationale behind the rules. It is assumed that all the rules in the RoA are motivated by the top-level safety goal that no collisions shall occur in the civil aerospace SoS. Figure 1 shows the first few stages of policy decomposition. All further examples refer to steps of the decomposition below these policy goals[1].

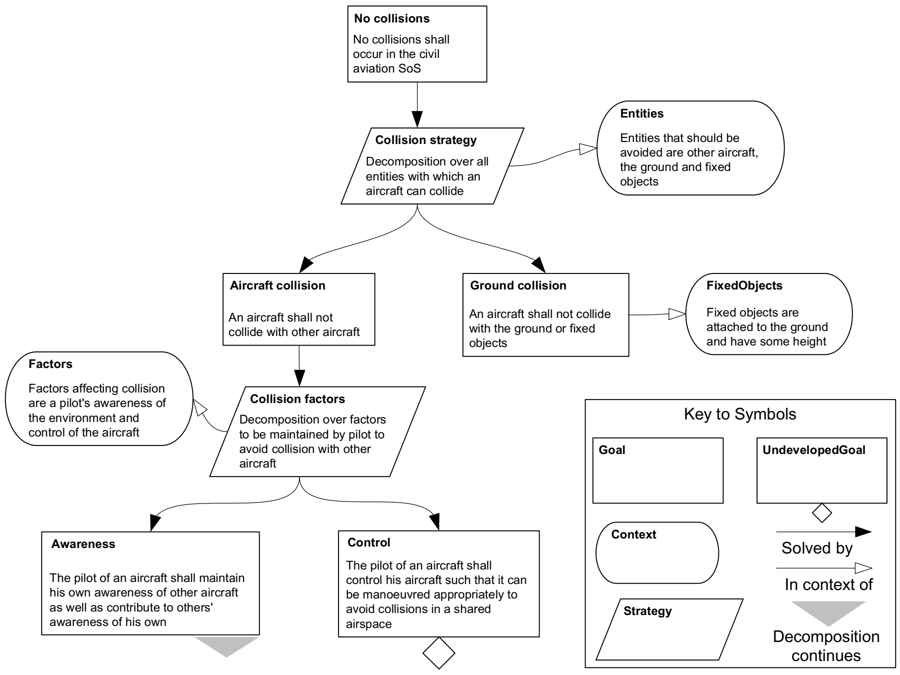

No collisions
No collisions shall occur in the civil aviation SoS
Collision strategy
Decomposition over all entities with which an aircraft can collide
Entities
Entities that should be avoided are other aircraft, the ground and fixed objects
Aircraft collision
An aircraft shall not collide with other aircraft
Ground collision
An aircraft shall not collide with the ground or fixed objects
FixedObjects
Fixed objects are attached to the ground and have some height
Factors
Factors affecting collision are a pilot's awareness of the environment and control of the aircraft
Collision factors
Decomposition over factors to be maintained by pilot to avoid collision with other aircraft
Awareness
The pilot of an aircraft shall maintain his own awareness of other aircraft as well as contribute to others' awareness of his own
Control
The pilot of an aircraft shall control his aircraft such that it can be manoeuvred appropriately to avoid collisions in a shared airspace
Key to Symbols
Goal
UndevelopedGoal
Context
Solved by
In context of
Strategy
Decomposition continues

Fig. 1. High Level Policy Decomposition for Rules of the Air

5.1 Agent-Based Decomposition

An agent-based decomposition concentrates on decomposing policies according to specific agents or groups of agents.

Agent Capabilities. Often it is desirable for a set of heterogeneous agents to adhere to a common policy. Clearly the way in which the policy must be broken down is dependent on the capability of said agent. This pattern is a specialisation of the case-based pattern, specifically each case represents a group of agents with a particular capability (or lack thereof).

Figure 2 demonstrates how a policy to fly at an altitude that minimises the chance of collision encounters can be decomposed over the ability of agents to

[1] Alphanumerical references in the goal decomposition — e.g. 17(a) — denote a particular rule in the Rules of the Air.

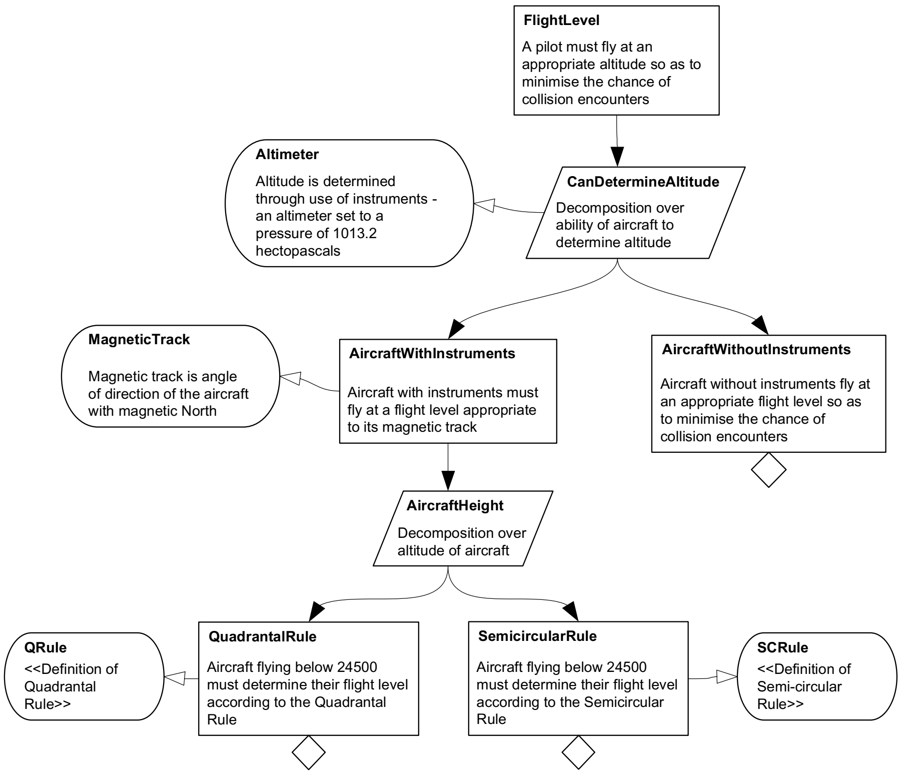

FlightLevel
A pilot must fly at an appropriate altitude so as to minimise the chance of collision encounters
Altimeter
Altitude is determined through use of instruments - an altimeter set to a pressure of 1013.2 hectopascals
CanDetermineAltitude
Decomposition over ability of aircraft to determine altitude
MagneticTrack
Magnetic track is angle of direction of the aircraft with magnetic North
AircraftWithInstruments
Aircraft with instruments must fly at a flight level appropriate to its magnetic track
AircraftWithoutInstruments
Aircraft without instruments fly at an appropriate flight level so as to minimise the chance of collision encounters
AircraftHeight
Decomposition over altitude of aircraft
QRule
<<Definition of Quadrantal Rule>>
QuadrantalRule
Aircraft flying below 24500 must determine their flight level according to the Quadrantal Rule
SemicircularRule
Aircraft flying below 24500 must determine their flight level according to the Semicircular Rule
SCRule
<<Definition of Semi-circular Rule>>

Fig. 2. Decomposition by Agent Capabilities

determine their altitude. There must be one policy for those systems able to determine their own flight level accurately (i.e. those with instruments), and one for those that cannot. Similarly, the policy could be decomposed over the ability of the agent type to modify their altitude — for instance, a glider may not climb in the same way as powered craft. Different ways of complying with the same policy must be devised for both types of agent.

In fact figure 2 also demonstrates how this decomposition pattern can be extended beyond agent capabilities to encompass all agent properties. The second strategy in the decomposition splits the policy according to the existing altitude of the aircraft. In this way it approaches the more general *case-based* pattern discussed later.

Agent Cooperation. This is a specialisation of the milestones pattern. However, in contrast with that pattern the milestones are assigned to different agents.

Consider figure 7. The policy that the conflict of right of way must be resolved when overtaking can be decomposed into the responsibilities of the two systems involved. One aircraft must cede to the other aircraft, which then has priority and is allowed to pass. These requirements on the agents are shown in figure 3

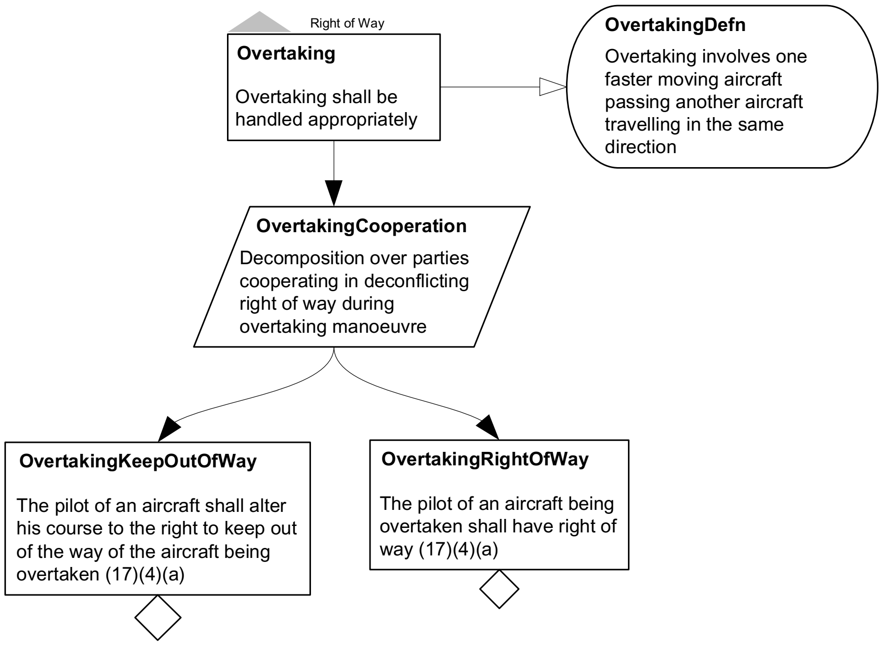

Right of Way
Overtaking
Overtaking shall be handled appropriately
OvertakingDefn
Overtaking involves one faster moving aircraft passing another aircraft travelling in the same direction
OvertakingCooperation
Decomposition over parties cooperating in deconflicting right of way during overtaking manoeuvre
OvertakingKeepOutOfWay
The pilot of an aircraft shall alter his course to the right to keep out of the way of the aircraft being overtaken (17)(4)(a)
OvertakingRightOfWay
The pilot of an aircraft being overtaken shall have right of way (17)(4)(a)

Fig. 3. Decomposition over Cooperating Agents

as two separate subgoals. The two policies are dependent on one another in that they represent the cooperation of two agents.

5.2 Milestone-Based Decomposition

A milestone-based decomposition attempts to decompose a policy goal by identifying an intermediate state to be achieved that contributes to the satisfaction of the policy goal. This is illustrated in figure 4. The milestone and the policy goal are temporally related; that is, the achievement of the milestone precedes the satisfaction of the final goal. The first subgoal states that the milestone be achieved, while the second subgoal defines a policy goal that can be achieved as a consequence of the milestone being achieved.

For example, a policy goal from the RoA identifies that to maintain good visibility pilots must not fly in poor weather conditions. This goal can be achieved by describing a milestone policy with two subgoals. The first subgoal requires the pilot first to become aware of the weather conditions through acquiring the latest weather forecast prior to take-off. A second subgoal requires that a pilot may not take-off if the forecast predicts bad weather (in the context of 'bad weather').

Variants of this particular example milestone occur frequently. An agent (human or machine) must first become aware of some state (be it troop movements or the state of the weather) whereupon some restriction on its actions is made

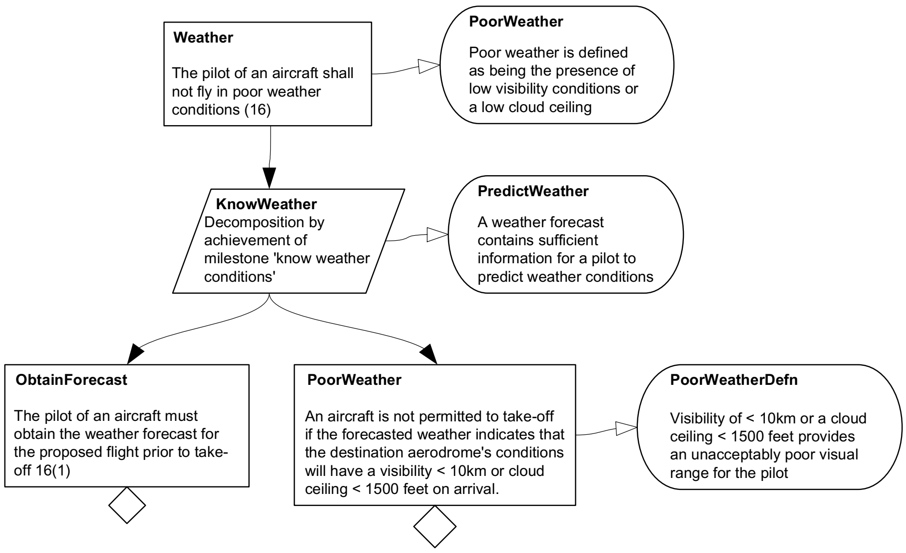

Weather
The pilot of an aircraft shall not fly in poor weather conditions (16)
PoorWeather
Poor weather is defined as being the presence of low visibility conditions or a low cloud ceiling
KnowWeather
Decomposition by achievement of milestone 'know weather conditions'
PredictWeather
A weather forecast contains sufficient information for a pilot to predict weather conditions
ObtainForecast
The pilot of an aircraft must obtain the weather forecast for the proposed flight prior to take-off 16(1)
PoorWeather
An aircraft is not permitted to take-off if the forecasted weather indicates that the destination aerodrome's conditions will have a visibility < 10km or cloud ceiling < 1500 feet on arrival.
PoorWeatherDefn
Visibility of < 10km or a cloud ceiling < 1500 feet provides an unacceptably poor visual range for the pilot

Fig. 4. Decomposition over Achievement of Milestone

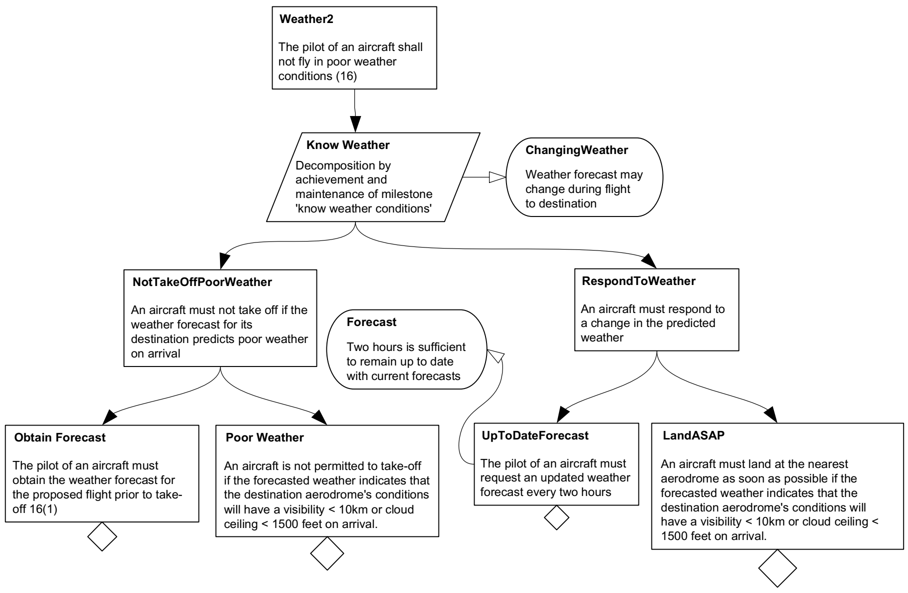

Weather2
The pilot of an aircraft shall not fly in poor weather conditions (16)
Know Weather
Decomposition by achievement and maintenance of milestone 'know weather conditions'
ChangingWeather
Weather forecast may change during flight to destination
NotTakeOffPoorWeather
An aircraft must not take off if the weather forecast for its destination predicts poor weather on arrival
Forecast
Two hours is sufficient to remain up to date with current forecasts
RespondToWeather
An aircraft must respond to a change in the predicted weather
Obtain Forecast
The pilot of an aircraft must obtain the weather forecast for the proposed flight prior to take-off 16(1)
Poor Weather
An aircraft is not permitted to take-off if the forecasted weather indicates that the destination aerodrome's conditions will have a visibility < 10km or cloud ceiling < 1500 feet on arrival.
UpToDateForecast
The pilot of an aircraft must request an updated weather forecast every two hours
LandASAP
An aircraft must land at the nearest aerodrome as soon as possible if the forecasted weather indicates that the destination aerodrome's conditions will have a visibility < 10km or cloud ceiling < 1500 feet on arrival.

Fig. 5. Decomposition over Achievement and Maintenance of Milestone

accordingly. This can be demonstrated by a standardised model of agent behaviour — such as OODA [9] — in that the observations made directly or indirectly (e.g. through third party information) affect the agent's actions.

All decomposition patterns are advisory; they guide the thoughts of the policy maker rather than constrict them. The decomposition process is not automatic, it is a creative process and the choice of a different pattern can lead to a different, but nonetheless viable policy.

A subtle variation on the milestone policy demonstrates this. Consider, instead of simply reaching the milestone once (treating it as a target), that the milestone were maintained in some way. This pattern leads to a subtly different decomposition of policy and hence affects the way the system operates. In this example the pilot would have not only to obtain the weather forecast but also to keep up to date with any changes (figure 5). This has non-trivial implications for the policy decomposition. It is too late to forbid take-off if the weather forecast predicts poor weather once the aircraft is in flight. The policy-maker then has a number of options for the pilot's behaviour: land at the next available opportunity, return to origin, or simply contact ATC and await instructions.

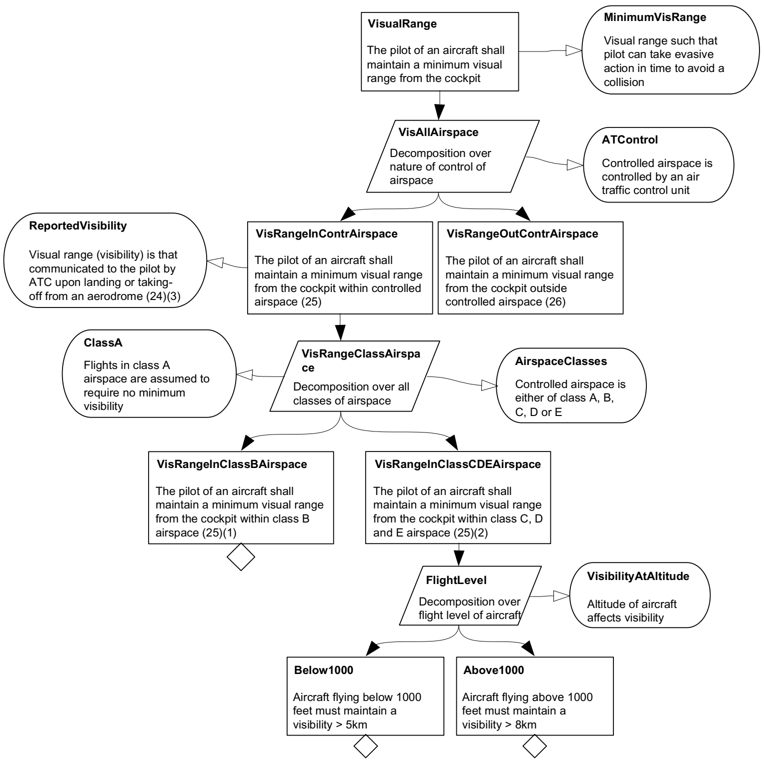

VisualRange
The pilot of an aircraft shall maintain a minimum visual range from the cockpit
MinimumVisRange
Visual range such that pilot can take evasive action in time to avoid a collision
VisAllAirspace
Decomposition over nature of control of airspace
ATControl
Controlled airspace is controlled by an air traffic control unit
ReportedVisibility
Visual range (visibility) is that communicated to the pilot by ATC upon landing or taking-off from an aerodrome (24)(3)
VisRangeInContrAirspace
The pilot of an aircraft shall maintain a minimum visual range from the cockpit within controlled airspace (25)
VisRangeOutContrAirspace
The pilot of an aircraft shall maintain a minimum visual range from the cockpit outside controlled airspace (26)
ClassA
Flights in class A airspace are assumed to require no minimum visibility
VisRangeClassAirspace
Decomposition over all classes of airspace
AirspaceClasses
Controlled airspace is either of class A, B, C, D or E
VisRangeInClassBAirspace
The pilot of an aircraft shall maintain a minimum visual range from the cockpit within class B airspace (25)(1)
VisRangeInClassCDEAirspace
The pilot of an aircraft shall maintain a minimum visual range from the cockpit within class C, D and E airspace (25)(2)
FlightLevel
Decomposition over flight level of aircraft
VisibilityAtAltitude
Altitude of aircraft affects visibility
Below1000
Aircraft flying below 1000 feet must maintain a visibility > 5km
Above1000
Aircraft flying above 1000 feet must maintain a visibility > 8km

Fig. 6. Decomposition into Disjoint Cases

5.3 Case-Based Decomposition

A case-based decomposition attempts to break the policy down into a number of cases, with each subgoal representing a case. Policy goals can be thought of as consisting of two parts: the conditions under which the policy must apply and the active part of the policy, i.e. the actions that are allowed or forbidden or states that are to be maintained etc. It would seem natural that a policy would 'always' apply, but this is deceptive. The conditions include not only temporal constraints but also the classes of systems the policy applies to as well as other restrictions. A policy goal with no conditions would be truly universal and apply always and to all things. The second part of the policy goal describes what the policy is to achieve. In decomposing policies both of these parts can be considered and broken down into simpler cases.

Decomposition into Condition Cases. The conditions in which a policy applies may be decomposed into specific cases of these conditions. These cases may overlap, i.e. the policies covering two or more cases can apply at the same time, or they may be totally disjoint.

Figure 6 shows an example of breaking down the fulfilment of a policy goal into a number of cases which do not overlap. Maintaining a sufficient visual range in all airspace can be broken down into those regions of airspace within ATC control and those outside. These two cases are obviously disjoint since there is no region of airspace that is not either uncontrolled or controlled by ATC. By applying the pattern again the policy that applies in controlled airspace can then be decomposed according to regions of airspace denoted with particular classes. In this case it must be asserted that the regions do not overlap, i.e. that there is no region that has more than one class assigned to it.

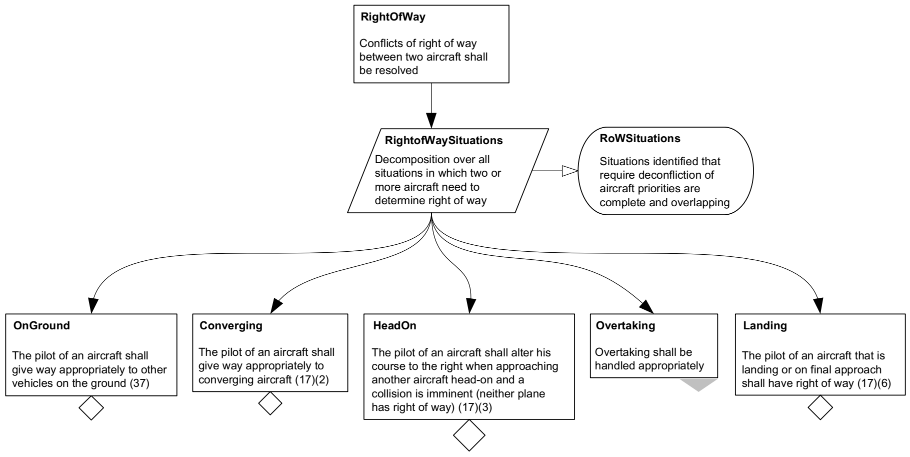

RightOfWay
Conflicts of right of way between two aircraft shall be resolved
RightofWaySituations
Decomposition over all situations in which two or more aircraft need to determine right of way
RoWSituations
Situations identified that require deconfliction of aircraft priorities are complete and overlapping
OnGround
The pilot of an aircraft shall give way appropriately to other vehicles on the ground (37)
Converging
The pilot of an aircraft shall give way appropriately to converging aircraft (17)(2)
HeadOn
The pilot of an aircraft shall alter his course to the right when approaching another aircraft head-on and a collision is imminent (neither plane has right of way) (17)(3)
Overtaking
Overtaking shall be handled appropriately
Landing
The pilot of an aircraft that is landing or on final approach shall have right of way (17)(6)

Fig. 7. Decomposition into Overlapping Cases

It is important to identify the type of case-based decomposition pattern because it has implications for how the child policies are formulated. Figure 7 illustrates the more tricky situation of decomposing the policy that right of way conflicts be resolved. One way to address this is by identifying all the cases in which a conflict can arise and generate a policy for each. Unfortunately it is not feasible to guarantee that the situations are completely disjoint. Where overlap between the cases occurs this implies that a resolution policy must be described for the intersection.

Decomposition into Active Cases. In a manner similar to identifying subcases of the conditions under which a policy applies, the active part of the policy can also be decomposed into cases. The policies covering the individual cases may be linked or convergent, where convergent means that any one of the policies individually fulfils the top-level policy and linked implies that all cases of the policy *interdependently* fulfil it [10]. Convergence does not necessarily mean that some of the policies are optional or that there is a choice. It means simply that each branch of the policy hierarchy below a policy goal independently fulfils this goal.

Figure 8 shows how the policy of maintaining a pilot's awareness can be broken down into two (linked) cases. On the one hand, the active case of observing allows a pilot to maintain awareness of the current local environment. Similarly the pilot must consider the case of passive observation, i.e. the fact that he is

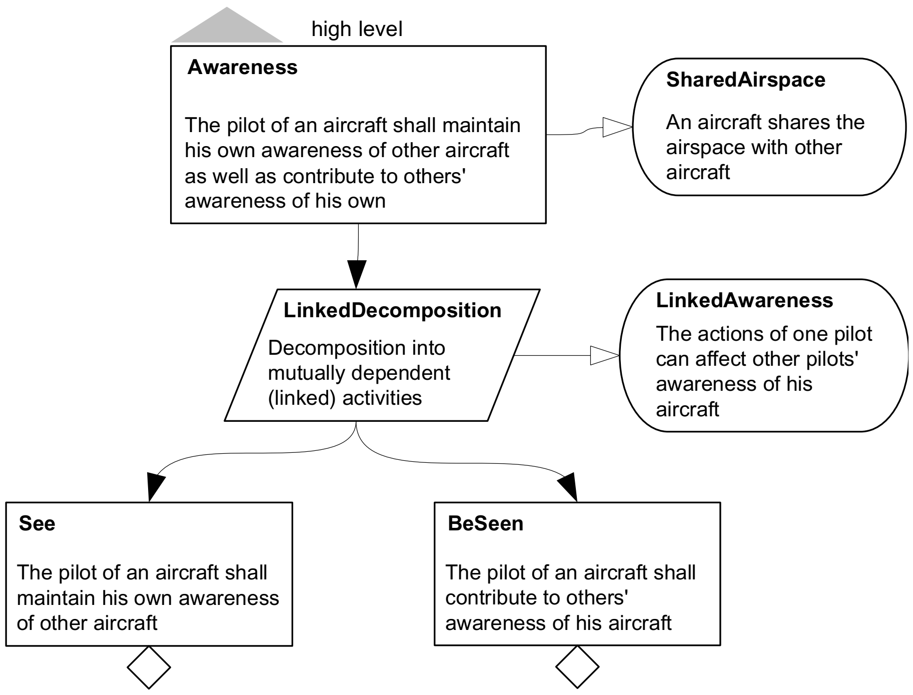

high level
Awareness
The pilot of an aircraft shall maintain his own awareness of other aircraft as well as contribute to others' awareness of his own
SharedAirspace
An aircraft shares the airspace with other aircraft
LinkedDecomposition
Decomposition into mutually dependent (linked) activities
LinkedAwareness
The actions of one pilot can affect other pilots' awareness of his aircraft
See
The pilot of an aircraft shall maintain his own awareness of other aircraft
BeSeen
The pilot of an aircraft shall contribute to others' awareness of his aircraft

Fig. 8. Decomposition into Linked Cases

being observed by other pilots. Policies facilitating both *seeing* and *being seen* are required in order for awareness in general to be maintained.

To deviate from the RoA briefly, taking an example from road traffic; a driver must signal the intention to turn or otherwise manoeuvre in good time. The policy (highway code) stipulates two possible ways of doing this: Either with mechanical indicators or using arm signals. These two policies are convergent in the sense that either one can reasonably indicate a driver's intention to turn. However, it should be clear that the use of arm signals provides a lower level of assurance [10] that the intention will be registered by other road users.

6 Problems of Defining a Resilient Policy

The very nature of the systems for which policy is being defined undermines the resilience of that policy. Systems of systems have dynamic structures, are distributed and consist of heterogeneous autonomous entities. It is these characteristics which necessitate the use of policy in the first place. However, they each present unique challenges.

Continual evolution of a dynamic SoS implies that systems are retired, replaced and upgraded and that the SoS has no well-defined 'end state'. This has implications for the resilience of policy because new systems can introduce new capabilities that break the context in which the original policy decisions were made. Similarly, systems that provide capabilities that were previously relied upon by policy can be withdrawn. The temptation is to create a policy that is liberal enough to accommodate such changes, whereas what is required is a process of recognising and systematically dealing with change [11].

The fact that systems are heterogeneous and that they change in this way means that any decomposition of policy that identifies a specific target system (e.g. a specific make and model of aircraft) will inevitably be wrong for future systems. To avoid such a 'brittle' policy implies that the lowest level of abstraction at which policy is expressed involves implementation by a target system.

Policy is therefore open to interpretation in the way it is implemented by autonomous systems. This can lead to various implementations and potential problems. Such misinterpretations must be, where possible, mitigated by an unambiguous policy. Unfortunately, unlike the KAOS patterns, safety policy is not afforded the luxury of an unambiguous refinement. It is clear that 'policy failure analysis' will need to be undertaken in order to predict the possible misinterpretations in implementing policy. Indeed the problem of preempting failures in decomposition has already been considered in [12].

So far, the issue of acceptable risk has remained implicit in our discussion. It can be implied that adhering to a defined set of policy objectives leads to an acceptable level of risk, while not following the policy leads to an unacceptable level of risk. However, such thinking masks how adherence, or non-adherence, to individual policy objectives contributes specifically to overall system risk. Other issues, such as the level of trust an agent has in its peer agents, is also masked.

Finally, because the systems are distributed means that coordinating policy distribution and adherence is not a trivial problem. In fact, given these characteristics, it would seem that the only reasonable way to evaluate policy is through simulation of a SoS. That is, by using policy to modify the simulated behaviour on a per-agent basis.

7 Related Work

This work draws on two areas of research: policy specification and goal decomposition. The use of policies to curb the behaviour of system entities is well established in the security and management domains. There are many notations used to express policies for controlling organisational complexity. For example, Ponder [13] is a language that attempts to present a unified approach to policy-based management and security. Ponder expresses policies in terms of authorisations and obligations in both positive and negative modalities. Whilst languages such as Ponder provide a means of expressing policy statements on agents, they do not deal well with the problems of expressing high-level policy objectives and their decomposition. Our work continues to look at how such policy languages can be integrated into policy decomposition.

There is also precedent in the area for the classification of policy into hierarchies of increasing abstraction [14,15,16]. It is suggested by Koch et al that a refinement of policy can be accomplished through the unambiguous mapping from one level of the hierarchy to the next. However, such an unambiguous mapping is not possible when considering safety policy goals for the reasons discussed in this paper. The approach presented in this paper is not strictly refinement (relying on deductive reasoning); instead it is a structured decomposition (relying on inductive reasoning). Other goal-directed decomposition approaches exist such as KAOS, TROPOS [17] and intent specifications [18]. However, none of these explicitly addresses the problems of systematising informal policy decomposition.

8 Further Work

The work presented in this paper provides a basis for resolving the problems of how to structure safety policy, however it is recognised that further work is necessary in a number of areas. Further evaluation of the use of patterns presented in this paper in defining new 'top-down' policy decompositions is required. Similarly, it is necessary to further define the context model of agents and its refinement, which is required to support the structuring and improved expression of policy goals. This improved expression of goals will aid in further work on detecting the potential for conflicts within and between policies in a multi-policy SoS. Finally, safety policy can be evaluated and improved by applying it to entities in a simulated SoS environment as discussed in a previous paper [19].

9 Conclusions

This paper has shown how it is possible to begin to structure policy using a pattern-based decomposition approach. It is desirable to be able to produce a safety policy, to which a system of systems can operate. This approach stresses the importance of recognising the contextual assumptions and strategies of decomposition often implicit in real-world policy documents. The Goal Structuring Notation, which is typically used to structure safety cases, was used to organise Rules of the Air into a hierarchy of policy goals at different levels of abstraction.

A number of patterns of decomposition based upon the KAOS tactics of agent-, case- and milestone-based refinement have been presented. These patterns have been adapted from the formal specification of KAOS since policy goals are not represented formally.

It has also been shown that the intrinsic characteristics of SoS leads to challenges for developing safety policy. Given an ostensibly perfect set of policy rules, the dynamic and heterogeneous nature of SoS means that interpretation and implementation of the policy may lead to 'failure' of the policy.

It is clear that future work must also entail analysing how policy is affected by changing scenarios. Simulation provides the basis for experimental validation of policy.

References

1. German Federal Bureau of Aircraft Accidents Investigation: Investigation report AX001-1-2/02 (2004)
2. Hall-May, M., Kelly, T.P.: Planes, trains and automobiles — an investigation into safety policy for systems of systems. To appear in Proceedings of the 23rd International System Safety Conference (2005)
3. Kelly, T.P.: Arguing Safety—A Systematic Approach to Managing Safety Cases. DPhil thesis, University of York, Heslington, York, YO10 5DD, UK (1998)
4. Darimont, R., van Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration. In: Proceedings of the 4th ACM Symposium on the Foundation of Software Engineering, San Francisco, California, USA (1996) 179–190
5. Polack, F., Stepney, S.: Emergent properties do not refine. In: Proceedings of the REFINE 2005 Workshop. ENTCS, Guildford, UK, Elsevier (2005)
6. Kelly, T.P., McDermid, J.A.: Safety case construction and reuse using patterns. In: Proceedings of the 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP '97), York, UK, Springer-Verlag (1997)
7. Darimont, R.: Process Support for Requirements Elaboration. PhD thesis, Université catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium (1995)
8. Allan, R., ed.: Air Navigation: The Order and the Regulations. third edn. Civil Aviation Authority (2003)
9. Boyd, J.R.: A discourse on winning and losing. Unpublished briefing, Air University Library, Maxwell AFB, Alabama, Report No. MU43947 (1987)
10. Weaver, R.A.: The Safety of Software — Constructing and Assuring Arguments. PhD thesis, University of York, Heslington, York, YO10 5DD, UK (2003)

11. Kelly, T.P., McDermid, J.A.: A systematic approach to safety case maintenance. In: Proceedings of the 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP '99). Volume 1698 of LNCS., Toulouse, France, Springer-Verlag (1999) 13–26
12. Armstrong, J., Paynter, S.: The deconstruction of safety arguments through adversarial counter-argument. In: Proceedings of the 23rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP '04). Volume 3219 of LNCS., Potsdam, Germany, Springer-Verlag (2004) 3–16
13. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: Ponder: A language for specifying security and management policies for distributed systems. Research Report DoC 2000/1, Imperial College, London (2000) http://www.doc.ic.ac.uk/deptechrep/DTR00-1.pdf.
14. Masullo, M.J., Calo, S.B.: Policy management: An architecture and approach. In: Proceedings of the 1st IEEE International Workshop on Systems Management, Los Angeles, California, USA, IEEE Computer Society Press (1993) 13–26
15. Koch, T., Krell, C., Krämer, B.: Policy definition language for automated management of distributed systems. In: Proceedings of the 2nd International Workshop on Systems Management, Toronto, Canada, IEEE Computer Society (1996) 55–64
16. Wies, R.: Using a classification of management policies for policy specification and policy transformation. In: Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management. Volume 4., Santa Barbara, California, USA, Chapman & Hall (1995) 44–56
17. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: Tropos: An agent-oriented software development methodology. Journal of Autonomous Agents and Multi-Agent Systems **8** (2004) 203–236
18. Leveson, N.G.: Intent specifications: An approach to building human-centered specifications. IEEE Transactions on Software Engineering **26** (2000) 15–35
19. Alexander, R., Hall-May, M., Despotou, G., Kelly, T.: Towards using simulation to evaluate safety policy for systems of systems. To appear in Proceedings of the 2nd International Workshop on Safety and Security in Multi-Agent Systems (SASEMAS '05) (2005)

Generalising Event Trees Using Bayesian Networks with a Case Study of Train Derailment

George Bearfield[1] and William Marsh[2]

[1] Atkins Rail, Euston Tower, 286 Euston Road, London, NW1 3AT, UK
george.bearfield@atkinsglobal.com
[2] Department of Computer Science, Queen Mary University of London, Mile End Road, London, E1 4NS, UK
william@dcs.qmul.ac.uk

Abstract. Event trees are a popular technique for modelling accidents in system safety analyses. Bayesian networks are a probabilistic modelling technique representing influences between uncertain variables. Although popular in expert systems, Bayesian networks are not used widely for safety. Using a train derailment case study, we show how an event tree can be viewed as a Bayesian network, making it clearer when one event affects a later one. Since this effect needs to be understood to construct an event tree correctly, we argue that the two notations should be used together. We then show how the Bayesian Network enables the factors that influence the outcome of events to be represented explicitly. In the case study, this allowed the train derailment model to be generalised and applied in more circumstances. Although the resulting model is no longer just an event tree, the familiar event tree notation remains useful.

1 Introduction

Event trees are used in quantified risk analysis to analyse possible accidents occurring as a consequence of hazardous events in a system. Event trees are often used together with fault trees, which analyse the causes of the hazardous event that initiates the accident sequence. Their origin goes back at least to the WASH-1400 reactor safety study in 1975 [1].

The most serious accident may be quite improbable, so an accurate assessment of the risk requires the probabilities of possible accident scenarios to be determined. The analysis of accidents must consider both the state of the system and of its environment when the hazardous event occurs. The analysis is made more difficult when the environment of a system is complex or variable.

Event trees model an accident as a sequence of events: this is an intuitive approach but it does not explicitly represent the state of the system and its environment, which influences the evolution of events. In this paper, we propose to address this limitation of event trees by using Bayesian Networks (BNs). We have applied this approach to a case study, adapting an existing event tree modelling a train derailment accident. The original author of the event tree was able to explain the system and environmental factors that had been considered when preparing the event tree, but which could not be included explicitly in it. Using a BN, these factors can be made explicit in the accident model, which can still be viewed as an event tree but is now more general with a single BN-based model taking the place of a set of related event trees.

We argue that the event tree and BN are complementary: an event tree can be translated into a BN allowing two views of the accident model, each view showing different properties of the model most clearly. The generalised model, with system and environmental factors that influence the events made explicit, is a BN but it can still be viewed using the event tree notation.

Event trees are supported by many software packages but are sufficiently simple to be created with standard tools such as a spreadsheet. Perhaps because of this, the notation used by different authors varies. Since we wish to translate between event trees and BNs, the first step, in Section 2, is a precise description of an event tree.

In Section 3, we introduce BNs and show how to translate an event tree into a BN. We first give a 'generic' translation based only on the number of events in the tree and then we give rules for simplifying the resulting BN. Section 4 introduces the case study and uses it to show that the combination of event trees and BNs allows a more general model of possible accidents. Conclusions and related work are in Section 5.

2 Event Trees

In this section, we give an informal but precise description of event trees, which will be the basis for the translation of event trees to BNs.

2.1 Events and Outcomes

The evolution of the system following the hazardous occurrence is divided into discrete *events*, starting from the *initiating event*. Each event has a finite set of outcomes; commonly there are just two outcomes – the event happens or does not happen – but a greater number of outcomes can be distinguished.

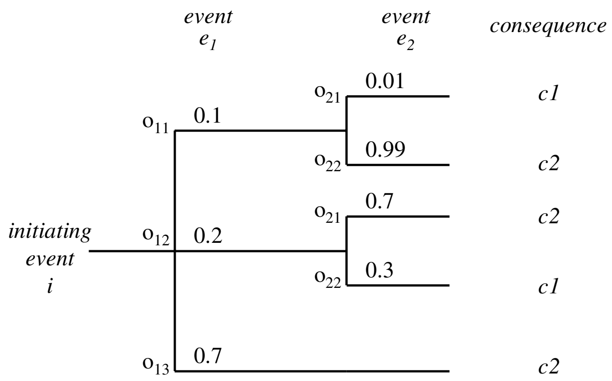


Fig. 1. An example event tree. There are two events: event e_1 has three possible outcomes o_{11}, o_{12} and o_{13} whereas e_2 has only two outcomes o_{21} and o_{22}. Two different consequences are distinguished c_1 and c_2; c_1 results both from the event sequence $i \rightarrow o_{11} \rightarrow o_{21}$ and from the event sequence $i \rightarrow o_{12} \rightarrow o_{22}$.

The events form a sequence in time: a tree of possible outcomes for all the events is constructed and the *consequence* or loss evaluated for each path through the tree. Some paths may be judged to lead to the same consequence. Fig. 1 shows an example event tree.

2.2 Probabilities and Consequence

The event tree specifies a logical combination of the event outcomes for each consequence. For the event tree in Fig. 1, the logical formulae for the consequences are $c_1 = (o_{11} \wedge o_{21}) \vee (o_{12} \wedge o_{22})$ and $c_2 = (o_{11} \wedge o_{22}) \vee (o_{12} \wedge o_{21}) \vee o_{13}$.

The probability of each consequence is calculated from the event probabilities, determined from data or experience. For example in Fig. 1, the probability of outcome o_{11} of e_1 event is 0.1. However, the probability of an outcome may depend on the outcomes of events earlier on the path: in Fig. 1 the probability of outcome o_{21} of event e_2 depends on the outcome of event e_1. The probabilities labelling the branches of the tree for e_2 are therefore conditional probabilities, in this example: $p(o_{21} \mid o_{11})$, $p(o_{22} \mid o_{11})$, $p(o_{21} \mid o_{12})$, and $p(o_{22} \mid o_{12})$.

The probabilities of the two consequences are calculated by multiplying the probabilities along each path and then adding the probabilities of paths leading to the same consequence. The calculation for Fig 1 is shown below.

Consequence	*Calculation*	*Result*
C_1	$0.1 \times 0.01 + 0.2 \times 0.3$	0.061
C_2	$0.1 \times 0.99 + 0.2 \times 0.7 + 0.7$	0.939

It is notable that the logical formulae for the consequences do not carry any information about how the outcome of one event is influenced by earlier events or even of how the events are ordered in time. The logical formulae are sufficient for combining the probabilities of event outcomes to give the consequence probabilities. On the other hand, understanding how the outcome of one event is influenced by earlier events is crucial for judging the event probabilities and the event tree shows only part of the information used during its construction:

- The time ordering of events shows the set of earlier events on which a probability may be conditioned; later events cannot influence the outcome of earlier events.
- However, some earlier events may have no influence and the event tree does not show what subset of the earlier events actually conditions each probabilities. Indeed, we have seen cases where inexperienced users of event trees are unaware that the probabilities attached to branches in an event tree are *conditional* probabilities at all.

In the example of Fig. 1, when event e_1 has outcome o_{13} the tree does not branch for the possible outcomes of event e_2. We refer to this as a *don't care* condition. There is more than one reason why the event tree may contain such a condition:

- Only one of the outcomes of e_2 is possible following the outcome of the earlier event.
- Both outcomes of e_2 are possible, but the consequence is the same for both.

It is important to note that the event tree does not distinguish between these reasons – there is no need to do so to calculate the consequence probabilities.

3 Translating an Event Tree to a Bayesian Network

In this section we first introduce BNs and describe a 'generic' representation of an event tree as a BN before showing how it can be simplified for a specific event tree.

3.1 Bayesian Networks

A BN [2] is a graph with a set of probability tables. The nodes of the graph represent uncertain variables and the arcs represent the causal relationships between the variables. The arcs are directed from 'parent' to 'child' with, conventionally, the parent as the cause and the child the effect. There is a probability table for each node, providing the probabilities of each state of the variable, for each combination of the states of parent variables. The model of cause is probabilistic rather than deterministic and this makes it possible to include factors that influence the frequency of events, but do not determine their occurrence.

Although the underlying theory (Bayesian probability) has been around for a long time, executing realistic models was only first made possible in the late 1980s using new algorithms. Methods for building large-scale BNs are even more recent [3] but it is only such work that has made it possible to apply BNs to the problems of systems engineering.

The RADAR group at QMUL, in collaboration with Agena Ltd, has built applications based on BNs that have shown the technology to be effective. Several such applications are for dependability assessment, notably the TRACS tool [4] used to assess vehicle reliability by QinetiQ (on behalf of the MOD) and a tool used by Philips to manage software quality [5].

3.2 A Generic Translation from ET to BN

Any event tree with three events e_1, e_2, and e_3 can be represented by the BN shown in Fig. 2. Two types of arc complete the network:

- Consequence arcs (shown as dotted lines in Fig. 2) connect each event node to the consequence node. This relationship is deterministic: the probability table for the consequence node encodes the logical relationship between the events and the consequences. (An example is shown in Fig. 5.)
- Causal arcs (shown as solid lines in Fig. 2) connect each event node to all events later in time. We say that e_1 influences the probability of (or, equivalently, is a causal factor for) event e_2.

We call this representation *generic* since the nodes and arcs depend only on the number of events. However, assuming that the BN is only used to determine the consequence probabilities (i.e. just as the event tree), some of the arcs may not be necessary allowing the BN to be simplified. In the next two sections we give rules for eliminating unnecessary arcs.

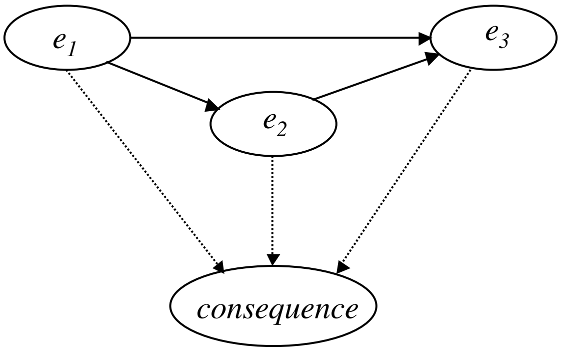


Fig. 2. Generic BN representation of an event tree. Nodes e_1, e_2, and e_3 represent the events; each node has a state for each outcome. The node *consequence* has a state for each of the consequences in the event tree.

3.3 Eliminating Consequence Arcs

The consequence arc from an event can be eliminated if the logical formulae for the consequences do not refer to any outcome of the event. Fig. 3 shows an example: the logical expression for c_1 is $(o_{11} \wedge o_{21}) \vee (o_{12} \wedge o_{21})$ but this can be simplified to o_{21}; since this expression (and the similar expression for c_2) includes only the outcomes of the e_2 event, the BN node e_1 is not needed as a parent of the *consequence* node. The set of consequence arcs is not determined by the branching structure of the event tree but by the assignment of consequences to each of the paths through the tree.

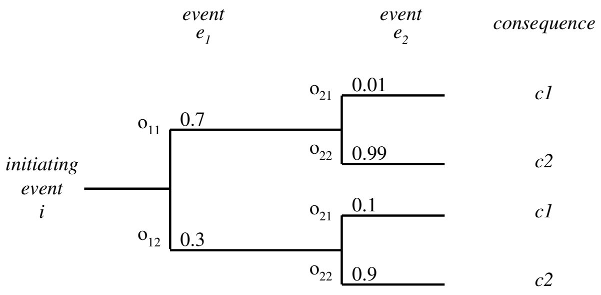


Fig. 3. Example of an event tree allowing a consequence arc to be eliminated, since e_2 determines the consequence whatever the outcome of the first event: the first event influences the relative probability of the two outcome of e_2 but does not change the consequence

3.4 Eliminating Causal Arcs

A causal arc to an event e_t from an earlier event e_f can be eliminated if and only if the probabilities labelling branches for event e_t do not depend on the outcome of event e_f. We can see this in the event tree: are the probabilities labelling an outcome o_{xy} the same on all branches for this outcome or do they differ? An example of this is shown in Fig. 4, where both branches for o_{21} have probability 0.1 and both branches for o_{22} have probability 0.9:

$$p(e_2 = o_{21} \mid e_1 = o_{11}) = p(e_2 = o_{21} \mid e_1 = o_{12}) = 0.1$$
$$p(e_2 = o_{22} \mid e_1 = o_{11}) = p(e_2 = o_{22} \mid e_1 = o_{12}) = 0.9$$

Because the probabilities of the outcome of event e_2 do not depend on the outcome of event e_1 no causal arc is needed from e_1 to e_2. More generally, if for all outcomes of e_t the probability $p(e_t \mid \ldots, e_f, \ldots)$ does not depend on the outcome of e_f (given the outcome of the other events) then the two events are 'conditionally independent' and the arc from e_f to e_t is not needed.

The complete BNs, including the probability tables, for the event trees in Figs 4 & 5, showing the two types of elimination, are given in Fig. 5.

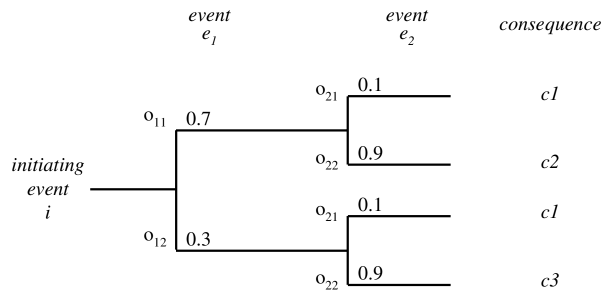


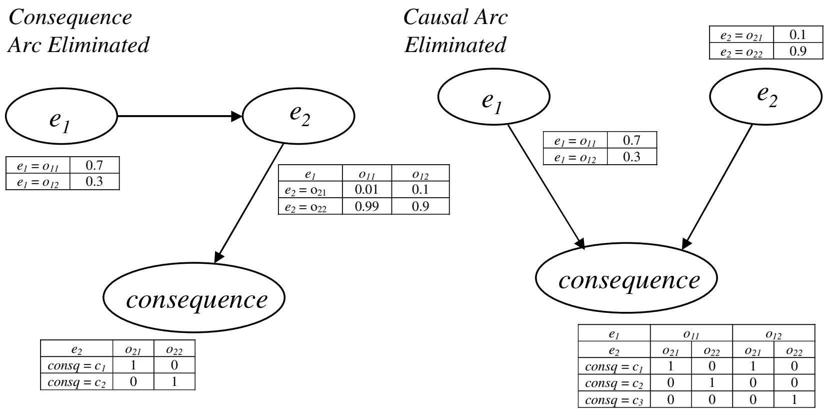
Fig 4. Example of an event tree allowing a causal arc to be eliminated: the probabilities of the two outcomes of event e_2 are the same whatever the outcome of event e_1. Note that this figure and Fig. 3 have the same shape but differ in the pattern of probabilities and consequences.

3.5 Handling 'Don't Care' Conditions

The event trees in Figs. 3 and 4 are both complete: a path exists for all possible combinations of outcomes of the two events. An event tree that is complete in this way includes all the probabilities needed to complete the node probability tables for the event nodes. However, this is not the case when there are *don't care* conditions in the event tree. In this section we show how the rules described above can be adapted for *don't care* conditions.

Consider the *don't care* branch in the event tree of Fig. 1: suppose that it is instead split into the two outcomes of event e_2, the first given probability α and the other 1-α. Any probability α could be used: since the two branches both lead to the same consequence (or set of consequences) the value chosen has no effect on the consequence probabilities. We are free to choose α to simplify the BN as far as possible, so we choose α to create conditional independence whenever this is possible.

This procedure produces the fewest causal arcs but it does not distinguish between the two reasons given at end of section 0 why a *don't care* condition may occur. This is satisfactory because the distinction doesn't affect the calculation of the consequence probabilities in either the event tree or the BN. However, by assuming that event outcomes are conditionally independent except when the probabilities shown in the event tree force the opposite conclusion we may have ignored causal relationships between events that really exist. If we use the BN model of the event sequence for other calculations we may need to add the causal arcs modelling these causal relationships to the BN. We could do this by determining the probabilities of the outcomes of *don't care* conditions and adding extra branches into the event tree. The resulting BN has some interesting properties but we do not need it to calculate consequence probabilities.



$e_1 = o_{11}$	0.7
$e_1 = o_{12}$	0.3

e_1	o_{11}	o_{12}
$e_2 = o_{21}$	0.01	0.1
$e_2 = o_{22}$	0.99	0.9

e_2	o_{21}	o_{22}
$consq = c_1$	1	0
$consq = c_2$	0	1

$e_2 = o_{21}$	0.1
$e_2 = o_{22}$	0.9

$e_1 = o_{11}$	0.7
$e_1 = o_{12}$	0.3

e_1	o_{11}		o_{12}	
e_2	o_{21}	o_{22}	o_{21}	o_{22}
$consq = c_1$	1	0	1	0
$consq = c_2$	0	1	0	0
$consq = c_3$	0	0	0	1

Fig. 5. Complete BNs for event trees of Figs. 4 & 5, showing the two types of arc elimination

3.6 Using a Hierarchy of Nodes for Consequence

Rather than having a single BN 'consequence' node with a probability table determined from the logical relationship between events and consequences, it is possible to represent this relationship using a hierarchy of nodes, determined from the event tree structure. A node can be introduced for each vertical line (representing a branch or decision point) in the event tree provided that more than two sequences lead from the branch. The parents of this node are the node representing the event and the nodes from the decision points to the right. Using a hierarchy of nodes has two potential advantages:

- more efficient propagation of the BN
- clearer representation (for the risk analyst) of the logical relationship between events and consequences.

We do not consider the efficiency of propagation further in this paper. In section 4.2, we assess whether the clarity of the model improves using this translation for a realistic event tree.

4 Why Use Bayesian Networks to Model Event Sequences

The previous section showed how to construct a BN equivalent to an event tree; however, if the two models are equivalent what purpose does the BN serve? We examine this using a case study of train derailment, which is introduced in section 4.1.

In the following sections, we first argue that an event tree and a BN provide complementary views of the relationship between events. Secondly, we show how an event tree expressed using a BN can usefully be generalised by making the factors influencing the evolution of events explicit, producing a more widely applicable model of the accident.

4.1 Case Study: Train Derailment

A 'Derailment Study' was carried out in 2001 as part of development studies for a proposed upgrade to an urban railway. The objective of the study was to quantify the risks to passengers and staff arising from derailment. This required the consequences of derailment to be analysed and event trees were constructed for this. Other models were used to analyse the frequency of derailment and, given the accident sequences, the likely toll of injuries. Since the ultimate aim was to ensure that risks were tolerable, some conservative assumptions were made.

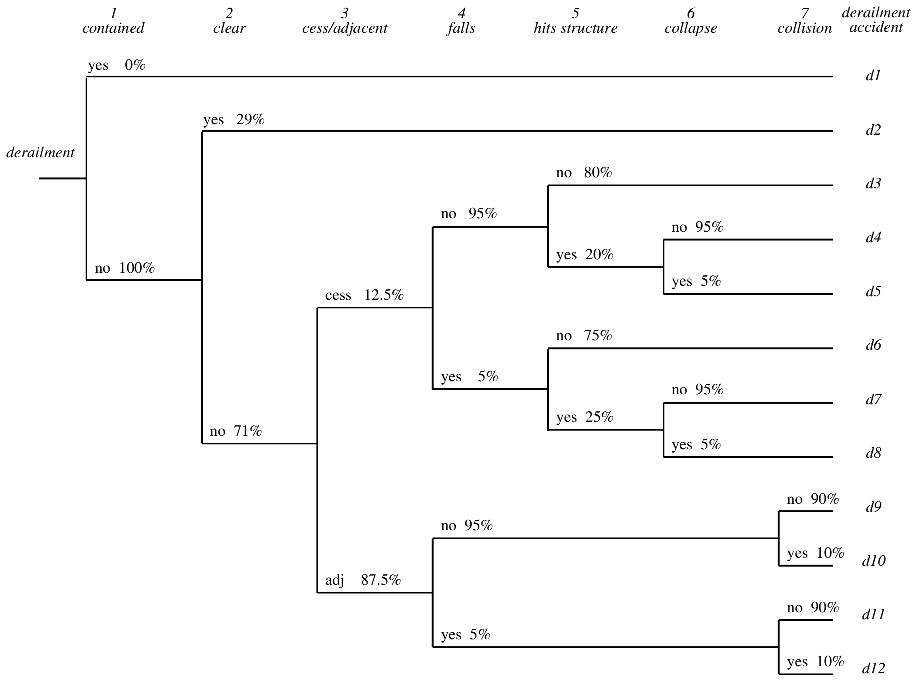


Fig. 6. An event tree from the 'Derailment Study' covering derailment in open track areas. The structure of the event tree, and the event probabilities, were adapted from a network-wide model by considering factors specific to the local circumstances.

The analysis used separate event trees for six different infrastructure areas, each with different characteristics including open track, in tunnels and on bridges. Here, we consider only derailments on areas of open track, which is track not in tunnels or carried on bridges. The analysis drew on a version of the 'Safety Risk Model' (SRM)

[6], which analyses the risk arising from different hazards using historic accident data and expert judgement for the UK rail network as a whole. The event trees for the derailment study used the structure of the SRM but had to be tailored to the local circumstances: for example, the maximum speed limit is 30 miles per hour, the trains are electric multiple units with third-rail electrification. The original author of the derailment study was available and assisted the authors with the case study.

The event tree for open track derailment is shown in Fig. 6. The events, all of which have only two outcomes, are described in Table 1. Twelve consequences or 'derailment accidents' are distinguished: for example 'd2' is 'minor derailment within clearance' and 'd7' is a 'major derailment to cess, striking line-side structure'. Given the frequency of the initiating 'derailment' event, the frequency of each accident can be calculated. The 'equivalent fatalities' for each accident are estimated by a separate method, which is not relevant here.

Table 1. Derailment Events

Event		*Description*
1	Derailment containments controls the train.	An extra raised 'containment' rail, if fitted, limits movement sideways.
2	The train maintains clearance.	The train remains within the lateral limits and does not overlap adjacent lines or obtrude beyond the edge of the track area.
3	Derails to cess or adjacent line.	The train can derail to either side of the track: derailing to the 'cess', or outside, may lead to a collision with a structure beside the line, while derailing to the 'adjacent' side brings a risk of colliding with another train.
4	One or more carriages fall on their side.	The carriages may remain upright or fall over.
5	Train hits a line-side structure.	The train hits a structure beside the line.
6	The train structure collapses.	Collision with a line-side structure causes the train structure to collapse.
7	Secondary collision with a passenger train.	A following or on-coming train collides with the derailed train.

4.2 Causality in the Event Sequence

Fig. 7 shows the BN generated for the event tree, using the algorithm described in section 3. Comparing the two notations – the BN of Fig. 7 and the event tree of Fig. 6 – we see that:

1. The logical combination of events leading to each accident is most clearly shown in the event tree.
2. The occurrence of conditional probabilities – arising from dependence between the events – is shown more clearly in the BN.

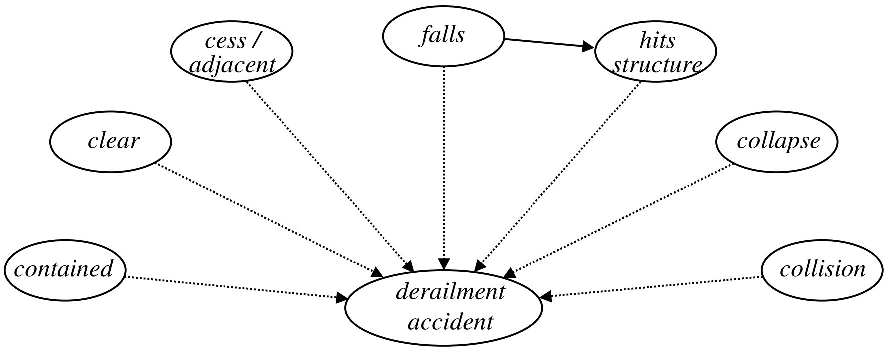


Fig. 7. Equivalent BN for open track derailment

The first point remains true even if the single ‘derailment accident’ node is replaced by a hierarchy of nodes as described in section 0, producing the BN shown in Fig. 8. Although this alternative translation may improve the efficiency of Bayesian propagation, the logical relationship between events and consequence is still more clearly shown in the original event tree.

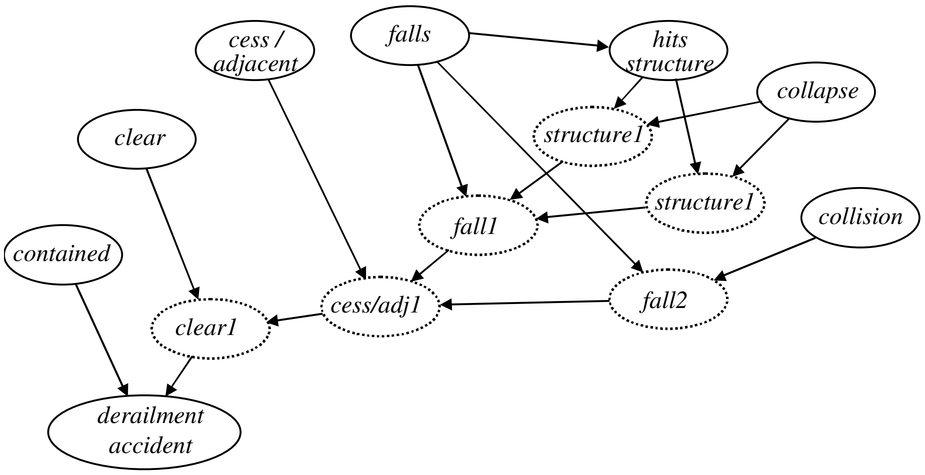


Fig. 8. Alternative translation of open track derailment event tree, using a hierarchy of nodes to encode the logical relationship between events and consequences

It may seem surprising that there is only a single causal arc – from ‘falls’ to ‘hits structure’ between the nodes representing events. This arc occurs because the probability *p(hits | falls = yes) ≠ p (hits | falls = no)*. For other events, the probability of each outcome is the same on all the branches. The absence of other causal arcs depends on our treatment of *don’t care* conditions. For example, a collision is only possible following a derailment to the adjacent side, but we do not need to represent this relationship by a causal arc since it is captured by the branching structure of the event tree. Since the two views of the event tree show different information most clearly, we propose to use them together: the BN view is used to ensure that conditional probabilities are handled correctly and the tree view is used for mapping event sequences to consequences. The BN can be shown without the consequence node and arcs, so this part of the BN can be chosen to optimise propagation.

4.3 Generalised Event Trees

As described above, the event tree was originally prepared from a network-wide event tree for derailment accidents. To be applied to an analysis in a specific location, the network-wide model has to be tailored. In this section, we show how a more general model can be represented as a BN, which can be tailored automatically.

The author of the event tree was asked to identify the conditions of the infrastructure and the operation of the railway that influence a derailment accident. Table 2 shows the conditions identified. The causal relationships between these conditions were then elicited together with the probability tables. Fig. 9 shows the resulting BN, with the consequence node and arcs omitted for clarity.

Table 2. Derailment Operating and Infrastructure Conditions

Conditions	**Description**
Fitted	Whether the derailment containment is fitted: Yes, No
Curvature	The curvature of the track: Severe, Mild, None
Number of tracks	The number of adjacent tracks: 2, 4
Track Speed	The running speed of the track (mph): 0-10, 10-30, 30-60, 60>
Derailment Speed	The speed of the derailment (mph): >15, <15
Lineside Density	The density of objects beside the line: High, Low
Lineside Type	The type of equipment beside the line: Fixed, Anchored
Density of Traffic	The traffic density: High, Low
Peak	The time of day when the incident occurs: Peak, Off peak
Passenger Loading	How full the coaches are: >50%, <50%
Crashworthiness	The crashworthiness of the train: High, Low
Rolling Stock	The type of rolling stock: High Speed Train, EMU

The relationships in the model are causal. For example, a train derailing on a tight curve will be more likely to exceed its clearances while one travelling in a straight line is more likely to maintain its clearances, as its momentum will tend to carry it forward in the direction of travel. The probability table for the event 'clear' (whether the train maintains clearance in a derailment) is:

Derailment Speed	> 15 mph			<15mph		
Curvature	None	Mild	Severe	None	Mild	Severe
Yes	0.75	0.6	0.29	0.9	0.7	0.4
No	0.25	0.4	0.71	0.1	0.3	0.6

The values 0.29 and 0.71 are taken from the original event tree (Fig. 6), since the circumstance of the original study were ‘Derailment Speed’ > 15 mph and severe track ‘Curvature’. The author of the original event trees judged the other probabilities: although the generalised model requires more such judgements they are similar to those needed to construct an event tree.

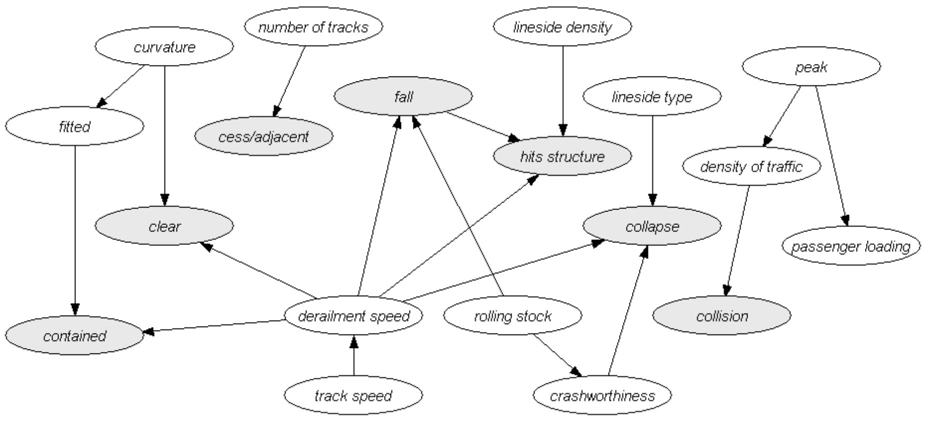


Fig. 9. Derailment BN generalised with the factors that determine the event probabilities. Event nodes are shaded; the consequence node and arcs are not shown.

The generalised model can be used to calculate the accident probabilities in different scenarios. We can compare the scenario in the original study (a dense urban line) with a scenario more typical of an inter-city line:

	Urban Scenario	**Inter-city Scenario**
Fitted	‘No’	‘No’
Curvature	‘Severe’	‘None’
Number of Tracks	4	2
Derailment Speed	‘> 15’ mph	‘> 15’ mph
Lineside Density	‘High’	‘Low’
Lineside Type	‘Anchored Equipment’	‘Fixed Equipment’
Rolling Stock	‘EMU’	‘High Speed Train’
Density of Traffic	‘High’	‘Low’

These data can be entered into the BN and new event probabilities calculated. The probabilities (relative to the probability of the initial derailment event) of the derailment accidents for the two scenarios are shown in Fig. 10. In the new scenario the less severe accidents are more likely: this results mainly from the absence of curvature. However, following the original study, we have considered only two possible derailment speed ranges and this should be re-examined before drawing any real conclusions. We also note that speed is a factor in the severities (equivalent fatalities) of the accidents, which are estimated using another method.

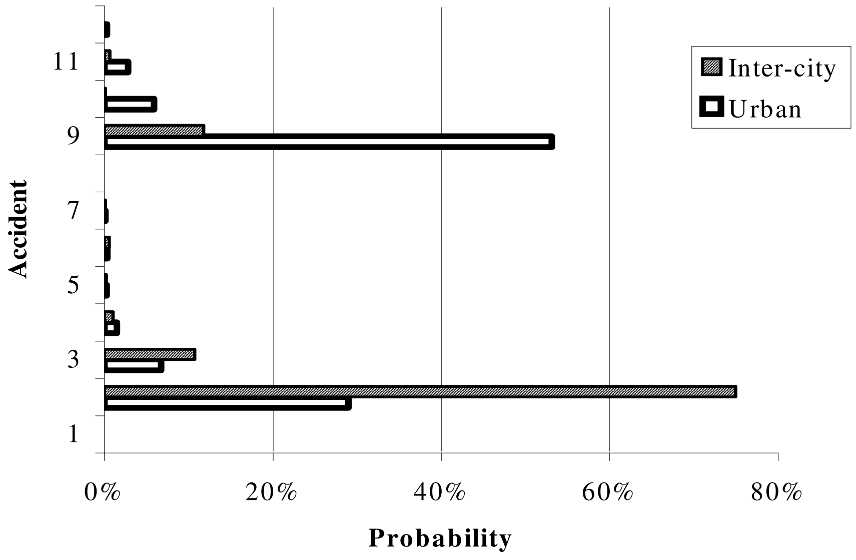


Fig. 10. Accident probabilities for two scenarios calculated using the BN. The 'urban' scenario is identical to the original derailment study giving the same probabilities as the event tree; the hypothetical scenario shows an example of the use of the generalised BN to adapt the accident analysis to different circumstances.

5 Discussion

5.1 Summary

We have shown how a BN can represent an event tree. The translation from BN to event tree is automatic (though we have not yet automated it) and reversible. We argue that the two notations are complementary and should be used together. The event tree shows the logical relationship of events, which is not shown clearly on the BN diagram where it is encoded in a probability table. On the other hand, the BN diagram shows clearly where event probabilities are conditioned on earlier events.

A greater advantage of using a BN is that the accident model can be generalised by including the conditions that influence the evolution of the events in the accident. This generalisation reverses the process used originally to analyse derailments in our case study, where an event tree for a specific location was developed from a network-wide model. The original author of the event tree remarked on the value of analysing causal influences on the events and was lead to re-examine some of the allocated probabilities.

It is advantageous to retain the familiar event tree notation when building the more general accident model. In the case study we were easily able to explain our approach to the author of the derailment event tree: only a short explanation of BNs was needed for this analyst to identify influencing factors and the causal relationships between them. Of course, generalising the accident model in the way we have shown is not automatic. A rigorous elicitation process is needed to understand the influences:

some remained unresolved in our case study, for example the influence of the train weight on the probability of the train falling over in a derailment. The process of judging probabilities for the BN, though time consuming, is similar to that required for building an event tree though potentially many more probabilities are needed.

The validity of the network-wide SRM rests on its use of historic accident data and it is desirable that an accident model for a specific location should have the same basis. At present, the SRM does not include influencing factors although the potential advantages of generalising it have been noted [7]. Clearly, further investigation of the cost-benefit of building such a model is needed.

5.2 Related Work

Others have used BNs to analyse risk. The SCORE project [8] has applied a BN to model accidents in an air-traffic control case study, based on a barrier model of accidents. In [9], an influence diagram is used to model the occurrence of rail breakage, also starting from a barrier model. In both cases the BN replaces the accident model used as a starting point – a barrier model rather than an event tree – rather than providing an alternative view as we have described.

Organisational and management causes of accidents are modelled using BNs in [10] and [8]. Organisational and management causes are examples of 'influencing factors' that could be included in our generalised event trees, so both are generalised representations of accidents, but without the connection to an underlying accident model such as an event tree, in the way we have proposed.

The SABINE emergency planning system [11] for accidents in nuclear power plants uses BNs. Part of this system is an accident diagnosis BN, derived from event trees constructed for level 2 PSA. Accident diagnosis requires back propagation from effects to causes and this is prevented by our simple and automatic treatment of *don't care* conditions (section 3.5) which may hide further causal relationships between event outcomes; rather than minimising the number of causal arcs in the BN, we could maximise it, including a causal arc wherever this is possible. We have not followed this approach because diagnosis is not required in our case study.

5.3 Further Work

The derailment study included six separate event trees for different areas of the infrastructure: we are examining how to merge these models. Existing software tools do not allow the event tree and BN views of the accident to be combined conveniently: we would like to investigate how to automate this in practice.

More fundamentally, some of the operating and infrastructure conditions also influence the causes of the initiating event: this is important because such factors introduce correlations between the probability of the initial event and the probabilities of different accident sequences. The present analysis does not capture such correlation and this could lead to an incorrect estimate of the risk. We plan to examine this in future.

Acknowledgement. We are grateful to Colin Howes of Atkins Rail for assistance with the case study, and to the referees' for their comments.

References

1. US Nuclear Regulatory Commission: Reactor Safety Study. WASH-1400, NUREG 75/014 (1975)
2. Jensen, F.V.: An Introduction to Bayesian Networks. UCL Press, London (1996).
3. Neil M., Fenton N.: Building Large Scale Bayesian Networks. Knowledge Engineering Review. 15(3) (2000) 257-284
4. Neil M., Fenton N., Forey S., Harris R.: Using Bayesian Belief Networks to Predict the Reliability of Military Vehicles. IEE Computing and Control Engineering J 12(1) (2001) 11-20.
5. Fenton, N.E., Krause, P., Neil, M.: Software Measurement: Uncertainty and Causal Modelling. IEEE Software 10(4) (2002), 116-122.
6. The Safety and Standards Directorate: Safety Risk Model (SRM), Report No. SP-RSK-3.1.3.8 (1999) Rail Safety and Standards Board, UK.
7. Bedford, T., French, S., Quigley, J.: Statistical Review Of The Safety Risk Model (WP1), Report T127 (WP1) (2004) Rail Safety and Standards Board, UK.
8. Neil, M., Malcolm B., Shaw R.: Modelling an Air Traffic Control Environment Using Bayesian Belief Networks. In: Proc. 21st Systems Safety Conference, Ottawa, ISBN 0-9721385-2-8.
9. Vatn, J., Svee, H. A.: Risk Based Approach to Determine Ultrasonic Inspection Frequencies in Railway Applications. Proceedings of the 22nd ESReDA Seminar, Madrid, Spain May 27-28, 2002
10. Roelen, A.L.C., Wever, R., Cooke, R.M., Lopuhaä, H.P., Hale, A.R., Goossens, L.H.J.: Aviation Causal Model using Bayesian Belief Nets to Quantify Management Influence. Safety and Reliability (Bedford & van Gelder eds.), Swets & Zeitlinger, Lisse, ISBN 90 5809 551 7, (2003) 1315-1320.
11. Zavisca M., Kahlert H., Khatib-Rahbar M., Grindon E., Ang M.: A Bayesian Network Approach to Accident Management and Estimation of Source Terms for Emergency Planning. In Proceedings of PSAM7/ESREL '04, Springer-Verlag (2004).

Control and Data Flow Testing on Function Block Diagrams

Eunkyoung Jee, Junbeom Yoo, and Sungdeok Cha

Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology(KAIST)
and AITrc/IIRTRC/SPIC,
373-1 Guseong-dong, Yuseong-gu, Daejeon, Republic of Korea
{ekjee, jbyoo, cha}@dependable.kaist.ac.kr

Abstract. As programmable logic controllers(PLCs) have been used in safety-critical applications, testing of PLC applications has become important. The previous PLC-based software testing technique generates intermediate code, such as C, from function block diagram(FBD) networks and uses the intermediate code for testing purposes. In this paper, we propose a direct testing technique on FBD without generating intermediate code. In order to test FBD, we define testing granularity in terms of function blocks and propose an algorithm that transforms an FBD network to a flow graph. We apply existing control and data flow testing coverage criteria to the flow graph in order to generate test cases. To demonstrate the effectiveness of the proposed method, we use a trip logic of BP(Bistable Processor) at RPS(Reactor Protection System) in DPPS(Digital Plant Protection System) which is currently being developed at KNICS[1] in Korea.

1 Introduction

Software testing is the act of exercising software with test cases for the purpose of finding failures [2]. Because failures of safety critical software can cause serious damage to life or property, testing of safety critical software has become an indispensable step required to assure software quality.

In the nuclear power plant control system, as existing analog systems have been replaced by digital systems controlled by software, testing of digital control systems has become more important. The control software is usually implemented on PLCs which are widely used to implement safety critical real-time systems. To test PLC applications, the characteristics of PLC programming languages should be considered. This work focuses on the FBD which is one of the most widely used standard PLC programming languages.

A PLC application implemented by FBD is automatically compiled to PLC machine code and executed on PLC. Testing of PLC machine code is difficult due to its complexity. Although the behavior of FBD is similar to the procedure or function of procedural program languages, there is no systematic way to apply software testing techniques to FBD. In previous cases[3], FBD testing has been done on intermediate C source code transformed from FBD networks. Although this method can test FBD networks at some level, it cannot be applied to FBD networks from which intermediate C code cannot be generated. Moreover, generating intermediate code leads to additional cost.

In this paper, we propose a direct cost-efficient testing method on FBD without generating intermediate code. We assume that the transformation process from FBD to PLC machine code has no errors. Because the transformation process has been validated for several decades by many PLC vendors, this assumption is reasonable. First, we define granularity of FBD testing. FBD is composed of network of function blocks. We define unit and module of FBD from the perspective of a function block network. In this paper, we focus on unit testing of FBD. To execute FBD unit testing, we propose an algorithm for the transformation of an FBD network to a flow graph. After generating a flow graph from an FBD network, we apply existing control and data flow testing strategies to the flow graph. To demonstrate the effectiveness of the proposed method, we use a trip logic of BP at DPPS RPS which is being currently developed at KNICS[1] in Korea.

The remainder of the paper is organized as follows: section 2 briefly introduces FBD and software testing, and section 3 defines granularity of FBD testing. In section 4, we propose an algorithm to transform an FBD network to a flow graph. We apply control and data flow testing strategies to the flow graph transformed from a real FBD example in section 5. Finally, conclusion and future works are described in section 6.

2 Background

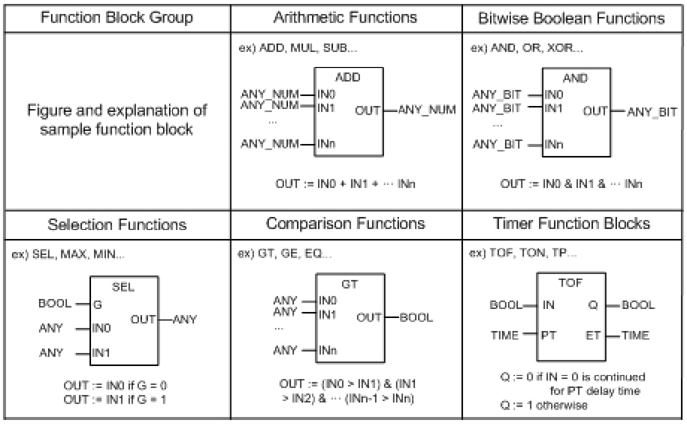
2.1 Function Block Diagram

A PLC[4] is an industrial computer widely used in control systems such as chemical processing systems, nuclear power plants or traffic control systems. A PLC is an integrated system that consists of a CPU, memory, and input- and output-points.

IEC 61131-3[5] identifies PLC programming languages, which includes Structured Text(ST), Function Block Diagram(FBD), Ladder Diagram(LD), Instruction List(IL), and Sequential Function Chart(SFC). FBD is one of the most widely used PLC languages. FBD is easy to understand and good for representing data flow between control blocks.

FBD represents system behaviors by means of signal flow among function blocks. Functions between input variables and output variables are configured by a network of function blocks in the form of a circuit. Function blocks figured by rectangles are connected by input variables on the left side and output variables on the right side. Function blocks are classified into several groups according to their functions.

Figure 1 shows several function block groups and an example function block of each group. RPS, which currently being developed at KNICS[1], is programmed with function blocks which belong to the five function block groups in figure 1.



Function Block Group	Arithmetic Functions	Bitwise Boolean Functions
Figure and explanation of sample function block	ex) ADD, MUL, SUB... ADD: ANY_NUM—IN0, ANY_NUM—IN1, ..., ANY_NUM—INn, OUT—ANY_NUM OUT := IN0 + IN1 + ⋯ INn	ex) AND, OR, XOR... AND: ANY_BIT—IN0, ANY_BIT—IN1, ..., ANY_BIT—INn, OUT—ANY_BIT OUT := IN0 & IN1 & ⋯ INn
Selection Functions	**Comparison Functions**	**Timer Function Blocks**

ex) SEL, MAX, MIN...
SEL: BOOL—G, ANY—IN0, ANY—IN1, OUT—ANY
OUT := IN0 if G = 0
OUT := IN1 if G = 1
	ex) GT, GE, EQ...
	GT: ANY—IN0, ANY—IN1, ..., ANY—INn, OUT—BOOL
	OUT := (IN0 > IN1) & (IN1 > IN2) & ⋯ (INn-1 > INn)
		ex) TOF, TON, TP...
		TOF: BOOL—IN, TIME—PT, Q—BOOL, ET—TIME
		Q := 0 if IN = 0 is continued for PT delay time
		Q := 1 otherwise

Fig. 1. Representative examples of function blocks

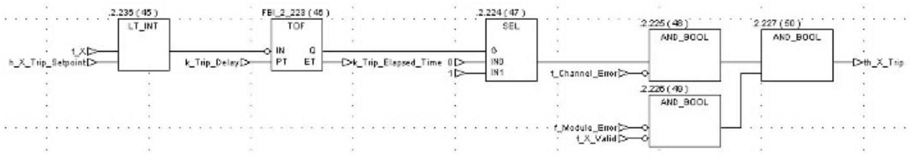


Fig. 2. An FBD program example

Figure 2 is an example of a function block network. The value of final output *th_X_Trip* is generated from the combined execution of several function blocks. The LT_INT function block in the leftmost position receives *f_X* and *h_X_Trip_Setpoint* as inputs and computes output. If *f_X* is less than *h_X_Trip_Setpoint*, it emits 1, else it emits 0. This output is inverted and used as an input to the TOF function block. The TOF function block is executed on the inverted output of previous function and *k_Trip_Delay* which is a constant for delay time. The output Q of the TOF function block is inputted into the next SEL function block. The output of the SEL function block enters the next SEL function block as input. If G is 0, the SEL function block selects and emits IN0 input, otherwise it selects and emits IN1 input. Finally, the AND_BOOL function block computes value of *th_X_Trip* which is the AND-ed combination of the output of SEL function block, inverted *f_Channel_Error*, inverted *f_Module_Error* and inverted *f_X_Valid*.

2.2 Software Testing

Software testing is the act of exercising software with test cases. There are two distinct goals of a test: to find failures, and to demonstrate correct execution [2]. Because it is hard to test all possible behaviors of software, the essence of software testing is to determine a set of test cases for the item being tested.

Each test case is the composition of inputs and expected output. Results of executing test cases are compared to expected outputs which are extracted from requirements specification.

There are two fundamental approaches to identifying test cases: functional and structural testing. Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range. The essential difference of structural testing with functional testing is that the implementation of the black box is known and used to identify test cases. Being able to see inside the black box allows the tester to identify test cases based on how the function is actually implemented [2].

This work applies the structural testing approach. Structural testing can be classified into two approaches. One is control flow testing that focuses on control flow in software. The other is data flow testing that focuses on the points at which variables receive values and the points at which these values are used. Both these two testing techniques are necessary and can be combined to complement each other.

Test coverage metrics are devices used to measure the extent to which a set of test cases covers a program [2]. It is used to decide if testing is executed adequately. Given a set of test cases, we execute test cases for the object program and determine how much of the program is covered. Through this, we can examine whether the test coverage criteria are satisfied.

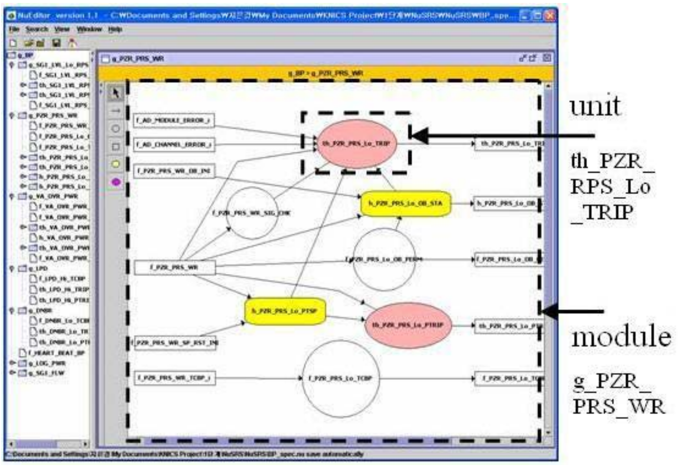
In this paper, we propose FBD testing technique in which we transform an FBD network to a flow graph and apply existing control and data flow testing strategies to the flow graph. We consider All-Nodes, All-Edges and All-Paths test coverage criteria in control flow testing, and All-Defs, All-Uses and All-DU-Paths test coverage criteria in data flow testing.

3 FBD Testing Granularity

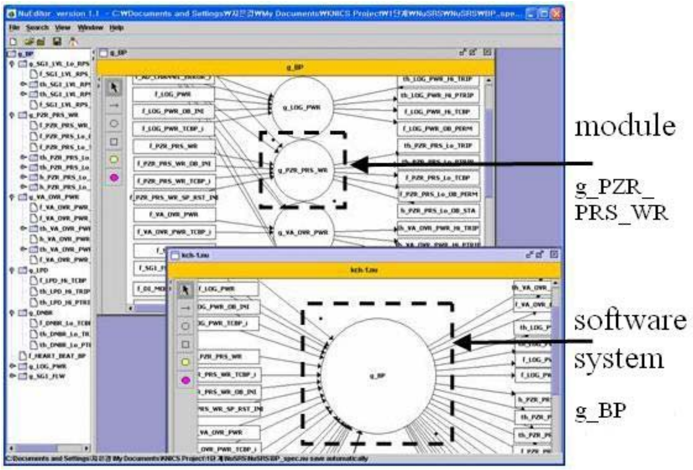
FBD is configured by a network of function blocks. Because established definitions of unit or module in procedural programming languages cannot be applied directly to FBD, we should define testing granularity from a view of function blocks.

If we define a unit as a function block in FBD, unit testing becomes unnecessary because we assumed that a function block always operates correctly. On the other hand, if we define a unit as a set of several function blocks, we have to consider the interaction between function blocks. This means that we should deal with integration testing issues. Therefore, proper definition for FBD unit and module is required.

We define a unit of FBD as 'a meaningful set of function blocks used to compute a primary output' [6]. The primary output is stored in the memory of the PLC for external output or internal use in other units. If the output variable is used just for programming conveniences, we do not consider it as a primary output. Figure 2 shows a part of KNICS RPS BP trip logic. It is the pre-trip set-point calculation part for manual reset variable set-point rising trip logic.



(a) FOD for *g_PZR_PRS_WR*



(b) FOD for *g_BP*

Fig. 3. FODs of KNICS RPS BP represented by NuEditor

This set of function blocks can be considered an individual unit because they perform the computation of an external output *th_X_Trip*.

We define a module as 'a set of units used to perform a meaningful function'. In KNICS RPS BP, each trip logic block can be defined as a module. Each module consists of several units.

In KNICS, NuSCR[7] is used to specify software requirements. NuSCR is a formal specification language that specializes in the nuclear power plant domain.

In NuSCR, the software system is specified by the combined use of FOD(Function Overview Diagram), FSM(Finite State Machine), TTS(Timed Transition System) and SDT(Structured Decision Table). We can get guidelines to divide the FBD into units and modules once the software requirement specification is written by the NuSCR.

Figure 3(a) shows an FOD for *g_PZR_PRS_WR*, a part of KNICS RPS BP, drawn by NuEditor[8], which is a tool that supports the NuSCR specification. An FOD represents relationship between various nodes using notation similar to data-flow diagram. FODs can represent complex hierarchical structures systematically by enabling nodes in higher level FOD to be refined in lower level FOD.

In KNICS, software requirements specified by NuSCR are implemented by FBD in the design step. In accordance with the definitions of unit and module in FBD, each node in figure 3(a) becomes an individual unit in FBD. An FOD in figure 3(a) corresponds to a module in FBD. This module is composed of 7 units with 6 inputs and 5 outputs. The higher level FOD for *g_PZR_PRS_WR* in figure 3(a) is the FOD for *g_BP* in the upper part of figure 3(b). The lower part of figure 3(b) shows an FOD with one node and several external inputs and outputs. It is the higher level FOD of the upper part FOD in figure 3(b). Figure 3(b) shows the highest level FOD in the system.

In FBD, a software system can be defined as a collection of all the modules of the system. A software system gets inputs from the outside of the system and emits outputs to the outside of the system. The highest level node *g_BP*, described in the lower part of figure 3(b), corresponds to the definition of a software system in FBD. Rectangles on the left side are external inputs, and rectangles on the right side are external outputs.

In this section, we defined the concept of units, modules and software systems in terms of function blocks for the testing of FBD. The division of units and modules of the FBD program can be easily identified using formal specification languages such as NuSCR.

4 Flow Graph Generation from FBD

To test FBD networks, we propose an algorithm to translate an FBD unit to a flow graph. If a flow graph can be generated from FBD, we can apply existing control and data flow testing techniques based on flow graphs to the flow graph[9,10,11]. Therefore, translation from FBD networks to flow graphs can be considered as the most fundamental and important process for FBD testing.

Figure 4 is an FBD unit which computes the value of *th_Prev_X_Trip*. This unit FBD is a part of the *g_PZR_PRS_WR* module presented in figure 3. It receives the pre-trip set-point value as input and determines the pre-trip value.

To translate the FBD unit to a flow graph, we have to understand the characteristics of the FBD execution. Every function block in FBD has its own execution order. Function blocks are executed sequentially in each scan cycle according to the corresponding execution order. In figure 4, the number inside parentheses on the top of function block is its execution order. For example,

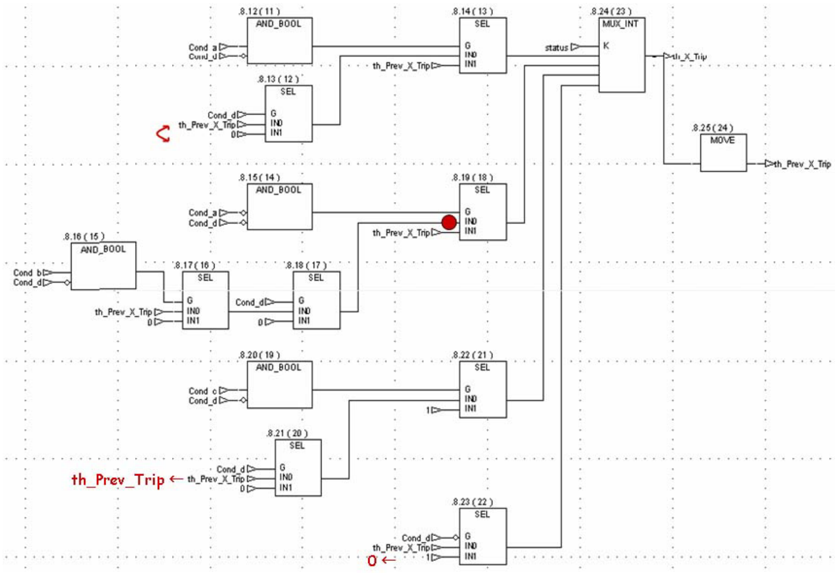


Fig. 4. FBD unit for *th_Prev_X_Trip*

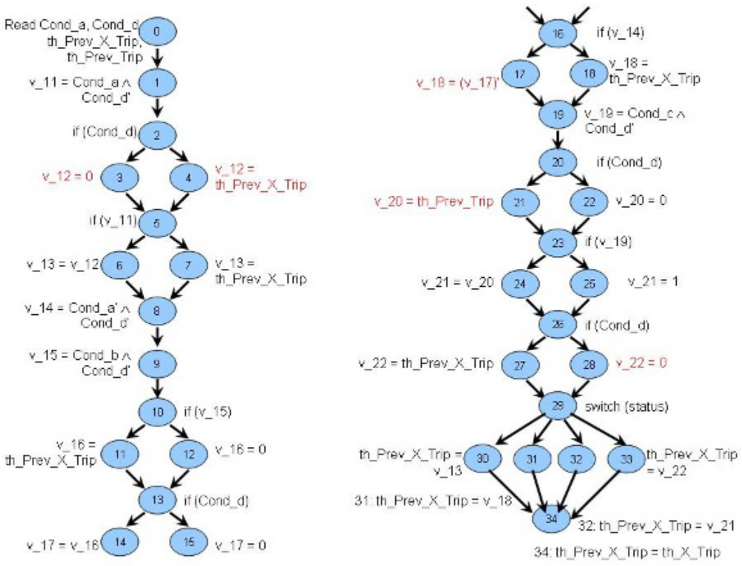


Fig. 5. Flow graph generated from FBD unit for *th_X_Pretrip*

in figure 4, the (14) AND_BOOL function block is executed first and the (24) MOVE function block is executed last. The translation process should reflect execution order characteristics of FBD correctly. Figure 5 is the flow graph translated from the FBD unit in figure 4. A flow graph is pictured by nodes and arrows. Each node is identified by a number assigned to it by the algorithm.

Figure 6 shows an algorithm which describes the process of translating an FBD to a flow graph. This algorithm does not cover all types of function blocks. According to this algorithm, we can generate nodes of the flow graph from the function blocks which belong to the arithmetic, bitwise Boolean, comparison, or selection group.

We describe the process of applying this algorithm to the FBD in figure 4. In lines 2–5, inputs needed to translate an FBD unit to a flow graph are described. Lines 15–19 generate a node which reads all input variables used in the FBD unit. The input variables of FBD in figure 4 are *Cond_a*, *Cond_b*, *Cond_c*, *Cond_d*, *status*, *th_Prev_X_Trip* and *th_Prev_Trip*. The node 0 in figure 5 is a node which reads these input variables. After inserting the first node into the flow graph, each function block is translated into nodes of the flow graph in the order of execution. The first executed function block in figure 4 is the (11) AND_BOOL. Because the AND_BOOL function block belongs to the bitwise Boolean function group, fb.group in line 23 is BITWISE_BOOLEAN and lines 28–33 are executed. Function blocks of function groups of ARITHMETIC, BITWISE_BOOLEAN and COMPARISON follow the same translation mechanism. Node 1 is created in lines 28–29 and variable v_11 is created in line 30.

In line 30, `outVar` := SetOutVariable(fb.executionNo, fb.outputVar) means that the output variable name is decided by SetOutVariable function. The (23) MUX_INT in figure 4 has the output variable *th_Prev_X_Trip* and the (24) MOVE has the output variable *th_Prev_Trip*. If an output variable of a function block

```
1:  procedure GenFlowGraphFromFBD (
2:         fbArray : FBArray;            { array of function blocks}
3:         startFbNum : Integer;         { execution order of the function
                                          block executed first }
4:         endFbNum : Integer;           { execution order of the function
                                          block executed last }
5:         inputVarList : StringList;    { input variables to FBD unit
                                          program} )

6:  var
7:      flowGraph : FlowGraph;
8:      fb : FunctionBlock;
9:      node : Node;
10:     childNodeList : NodeList;
11:     curNodeNum : Integer;
12:     outVar, contentString : String;

13: begin { GenFlowGraphFromFBDs }
14:    { First node 0 is generated. First node has content that read all
         input variables of the unit }
15:     curNodeNum := 0;
16:     node := CreateNode(curNodeNum);
17:     contentString := MakeContent(READ, inputVarList);
18:     node.content := contentString;
19:     InsertNode(flowGraph, node);
```

```
    { Translate function blocks to nodes of flow graph
      from first executed one to last executed one }
    for curFbIndex := startFbNum to endFbNum
       fb := fbArray[curFbIndex];
       switch(fb.group)
         case ARITHMETIC :
         case BITWISE_BOOLEAN :
         case COMPARISON :
           { A node whose content is assigning fb.outputSpec to
             outVar is generated and inserted to flow graph }
           curNodeNum := curNodeNum + 1;
           node := CreateNode(curNodeNum);
           outVar := SetOutVariable( fb.executionNo,
                                     fb.outputVar);
           contentString := MakeContent( ASSIGN,outVar,
                                     fb.outputSpec);
           node.content := contentString;
           InsertNode(flowGraph, node);
         case SELECTION :
           { Some of function blocks are translated  to
             if-then- else or switch structure }
           if fb.name = SEL or
              fb.name contain MUX then
              { generate condition node }
              curNodeNum := curNodeNum + 1;
              node := CreateNode(curNodeNum);
              contentString := MakeContent( IF, fb.condVar);
              node.content := contentString;
              InsertNode(flowGraph, node);
              { identify output variable of function block }
              outVar := SetOutVariable( fb.executionNo,
                                        fb.outputVar);
              { generate child nodes according to the condition }
              for i=0 to fb.inputNum-1
                 curNodeNum := curNodeNum + 1;
                 node := CreateNode(curNodeNum);
                 contentString := MakeContent( ASSIGN, outVar,
                                               fb.in[i]);
                 node.content := contentString;
                 AddNode(childNodeList, node);
              end { for_i }
              InsertMultipleNodes(flowGraph, childNodeList);
            else
              curNodeNum := curNodeNum + 1;
              node := CreateNode(curNodeNum);
              outVar := SetOutVariable ( fb.executionNo,
                                         fb.outputVar);
              contentString := MakeContent( ASSIGN, outVar,
                                            fb.outputSpec);
              node.content := contentString;
              InsertNode(flowGraph, node);
           end { if fb.name }
         case default : break;
      end { switch }
   end { for curFbIndex }
   output flowGraph;
 end { GenFlowGraphFromFBD }
```

Fig. 6. An algorithm for generating a flow graph from an FBD unit program

is identified in a fashion similar to these examples, the output variable name of the function block will be assigned to the outVar variable. If output variable of the function block is not identified, we generate new variable, such as v_11 or v_12, and assign it to outVar.

In line 31, MakeContent(ASSIGN, outVar, fb.outputSpec) generates node content where fb.outputSpec is assigned to outVar, which is then assigned to contentString. The (11) AND_BOOL function block receives two inputs *Cond_a* and *Cond_d* and executes an AND operation. In this case, MakeContent can generate a statement such as "v_11 = $Cond\_a \wedge Cond\_d$", which becomes the content of node 1 in line 32. Because the (11) AND_BOOL function block does not have divided control flow, it becomes a single node in the flow graph.

The (12) SEL function block outputs *th_Prev_X_Trip* or 0 according to the conditional input *Cond_d*. In this case, the SEL function block has two control branches according to the input, so that it is translated into an if-then-else structure, like nodes 2,3 and 4 in figure 5. Lines 37–42 describe the process of making node 2 and lines 43–53 describe the process for node 3 and 4. The (23) MUX_INT function block is translated into a structure with multiple control flows. Applying lines 34–53 to the (23) MUX_INT function block results in nodes 29–34 in figure 5.

5 FBD Unit Testing

5.1 Control Flow Testing

After transforming the FBD unit program to a flow graph, we select proper test coverage criteria and generate satisfying test cases.

Control flow testing coverage criteria include All-nodes, All-edges and All-path [9]. To satisfy the All-nodes coverage criterion, all nodes in flow graph should be executed at least once by test cases. All-edges coverage criterion requires that all edges in flow graph should be executed at least once. All-edges test coverage criterion subsumes All-nodes coverage criterion because test cases by which all edges are visited are sure to visit all nodes. All-paths test coverage criterion requires that every possible complete path in the program should be tested. All-paths coverage subsumes All-edges coverage, and therefore also subsumes All-nodes coverage. It is difficult to satisfy All-paths test coverage criterion due to its stringency.

Table 1 shows test cases that satisfy the All-edges test coverage criteria for the flow graph in figure 5. The six columns *Cond_a*, *Cond_b*, *Cond_c*, *Cond_d*, *status* and *th_Prev_X_Trip* represent six input variables for the *th_X_Pretrip* unit program. The rightmost column is the expected output for the *th_Prev_X_Trip* variable which is the final output of this unit program. After generating test cases, we get actual output by executing test cases through the flow graph, and then compare the actual output to the expected output. If two values are different, it means that the program has some errors.

5.2 Data Flow Testing

Data flow testing refers to forms of structural testing that focus on the points where variables receive values and the points where these values are used (or

Table 1. Test cases satisfying All-edges test coverage criterion

Test Case	*Cond_a*	*Cond_b*	*Cond_c*	*Cond_d*	*status*	*th_Prev_X_Trip*	Expected Output
CT1	1	0	1	0	0	1	1
CT2	1	1	0	1	1	0	0
CT3	0	0	0	0	2	1	1
CT4	0	1	1	1	3	0	1

referenced) [2]. Node n is a defining node of the variable v if and only if the value of the variable v is defined at the statement fragment corresponding to node n. There are two forms of definition nodes: definition by input and definition by assignment. Node n is a usage node of the variable v if and only if the value of the variable v is used at the statement fragment corresponding to node n. Path from a definition node to a usage node is du-path. A definition-clear path with respect to a variable v is du-path such that no other node in the path is a defining node of v [10]. For example, the content of node 1 in figure 5 is 'v_11 = *Cond_a* $\wedge$ *Cond_d*'. Node 1 is a definition node with respect to variable v_11, and a usage node with respect to variable *Cond_a* and *Cond_d*.

To apply data flow testing to the program, we first identify definition and usage nodes for all the program variables. We also identify du-paths with respect to each variable, and then apply All-Defs, All-Uses or All-DU-paths test coverage criteria. If T is a set of paths in the flow graph, the set T satisfies the All-Defs criterion for the program if and only if for every variable v, T contains definition-clear paths from every defining node of v to a use of v. All-Uses and All-DU-paths coverage criteria are defined similarly [11].

There are two types of variables in the flow graph generated from an FBD unit. One is input and output variables of the FBD, and the other is temporary variables. Temporary variables typed as 'v_number' store outputs of function blocks and are created during the transformation process. In data flow testing for FBD, we have to consider both types of variables. We need to identify definition and usage nodes for all variables and extract du-path information for each variable.

In comparison to other data flow testing, FBD data flow testing has several defining characteristics. When transforming the FBD to a flow graph, we created the first node of the flow graph with the content of reading all input variables of the unit FBD program. The first node of flow graph becomes a definition node for all input variables. For example, the FBD unit program in figure 4 has six input and output variables - *Cond_a*, *Cond_b*, *Cond_c*, *Cond_d*, *th_Prev_X_Trip*, *th_Prev_Trip* and *status*. These variables are all read in the first node 0. Thus, node 0 becomes the definition node for all input and output variables.

Temporary variables and an output variable of a unit FBD have definition nodes only by assignment; they have no definitions by input. This is one of the characteristics of FBD data flow testing. The variable *th_X_Trip*, used as an output variable at (23) MUX_INT in figure 4, is defined by assignment at node 30–33 of the flow graph in figure 5. The variable *th_X_Trip* has four definition nodes - node 30, 31, 32 and 33.

The du-paths for temporary variables are all definition-clear paths because each temporary variable has only one definition node and one usage node in a path. Moreover, except for the timer function group, FBDs have no loop construction; therefore we do not have to consider loop structure when generating test cases.

Table 2 shows test cases satisfying All-DU-paths test coverage criteria. They are generated with du-path information for all variables in the flow graph.

Table 2. Test cases satisfying All-DU-paths test coverage criteria

Test Case	*Cond_a*	*Cond_b*	*Cond_c*	*Cond_d*	*status*	*th_Prev_X_Trip*	Expected Output
DT1	0	0	0	0	0	1	1
DT2	1	0	0	1	1	1	0
DT3	1	1	0	0	0	1	1
DT4	0	0	0	0	1	0	0
DT5	0	1	0	0	2	0	0
DT6	1	1	1	0	2	0	1
DT7	1	1	1	0	3	0	0
DT8	0	0	0	1	3	1	1

5.3 Case Study

We applied the proposed approach to BP trip logic of DPPS RPS, which is being developed in KNICS. This section explains how we can find various errors in FBD program using the proposed FBD testing method. First, we seeded four different errors to the *th_Prev_X_Trip* FBD unit in figure 4. All these errors frequently occur in FBD programming. More errors occurring in FBD programming are explained and classified in [12]. The seeded faults were all found by test cases satisfying All-edges test coverage criteria in table 1. Test cases satisfying All-DU-paths test coverage criteria in table 2 could also find all seeded faults.

– *Case 1 (Switched input)*: One of the frequently occurring mistakes in FBD programming is switched input. While change of the input order in AND_BOOL function blocks is not a problem, switched input in SEL, MUX, or GE_INT function blocks can cause serious errors. For these kinds of function blocks, the correct order of inputs is important. We reversed inputs of the (12) SEL function block. We assign IN0 '0' instead of '*th_Prev_X_Trip*' and assign IN1 '*th_Prev_X_Trip*' instead of '0'. In control flow testing, this fault was found by the CT1 test case in table 1. In data flow testing, it was found by the DT1 test case in table 2. We found that the expected output of *th_Prev_X_Trip* is 1, but actual output of executing the test case is a different value, namely 0.
– *Case 2 (Misused inverter)*: The inverter, drawn by small circle, is often added in unnecessary positions or omitted in necessary positions. Misused inverters are also one of the frequently occurring errors in FBD programming. We

inserted an unnecessary inverter to IN0 input of the (18) SEL function block. In control flow testing, this fault was found by the CT2 test case in table 1. In data flow testing, it was found by the DT2 test case in table 2.
- *Case 3 (Incorrect variable)*: Variable names are often written incorrectly. Incorrect variable names result in wrong value assignments or computations. We wrote an input of (20) SEL as *th_Prev_Trip* instead of *th_Prev_X_Trip*. This fault was found by the CT3 test case of table 1 in control flow testing and by the DT5 test case of table 2 in data flow testing.
- *Case 4 (Incorrect constant)*: Another case of input errors is incorrect constant. We changed the original input of IN1 in the (22) SEL from 1 to 0. Wrong descriptions between 0 and 1 occur frequently. This fault was found by the CT4 test case of table 1 in control flow testing and by the DT8 test case of table 2 in data flow testing.

6 Conclusion

We proposed a direct testing technique on FBD without generating intermediate source code. A previous approach for FBD testing generates intermediate C code and performs testing on the intermediate code. Because the previous approach requires generating intermediate C code, it cannot be applied to FBD which do not generate intermediate C code. Moreover, generating intermediate code leads to additional cost. We proposed a cost-efficient testing method for FBD by applying testing strategies to FBD directly.

We defined unit and module of FBD in the view of function blocks and proposed an algorithm for translating an FBD unit to a flow graph. After generating a flow graph from an FBD unit, we applied existing testing techniques to the flow graph. In control flow testing, we generated test cases satisfying the All-edges test coverage criteria. We also generated test cases satisfying the All-DU-paths coverage criteria in data flow testing. To demonstrate the effectiveness of the proposed method, we used a trip logic of BP in DPPS RPS which is currently being developed at KNICS [1] in Korea. We seeded frequently occurring errors into the example FBD. We were able to find all seeded faults by the test cases generated by the proposed approach.

We have a plan to take timer function blocks into consideration. The transformation algorithm from an FBD unit to a flow graph has to be supplemented in order to cover FBD units with timer function block where we have to deal with time and state as well as input variables. Integration testing of FBD which focuses on interfaces and interactions between tested units should also be considered.

Acknowledgments

This work was partially supported by the Korea Science and Engineering Foundation(KOSEF) through the Advanced Information Technology Research Center (AITrc). This work was also partially supported by the Information Technology

Research Center(ITRC), Software Process Improvement Center(SPIC) and Internet Intrusion Response Technology Research Center(IIRTRC).

References

1. KNICS, Korea Nuclear Instrumentation and Control System Research and Development Center,http://www.knics.re.kr/english/eindex.html.
2. Paul C. Jorgensen, "Software testing: a craftsman's approach", CRC Press, 1995
3. http://www.framatome-anp.com
4. A. Mader, "A Classification of PLC Models and Applications", In Proc. WODES 2000: 5th Workshop on Discrete Event Systems, August 21-23, Gent, Belgium, 2000.
5. IEC, International Standard for Programmable Controllers: Programming Languages (Part 3), 1993.
6. J. Yoo, S. Park, H. Bang, T. Kim, S. Cha, "Direct Control Flow Testing on Function Block Diagrams," The 6th International Topical Meeting on Nuclear Reactor Thermal Hydraulics, Operations and Safety (NUTHOS-6), Nara, JAPAN, Oct. 4-8, 2004
7. J. Yoo, T. Kim, S. Cha, J-S. Lee, H.S. Son, "A Formal Software Requirements Specification Method for Digital Nuclear Plants Protection Systems", Journal of Systems and Software, Vol.74, No.1, pp73-83, 2005.
8. J. Cho, J. Yoo, S. Cha, "NuEditor - A Tool Suite for Specification and Verification of NuSCR", In proc. Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004), pp298-304, LA, USA, May 5-7, 2004.
9. E. F. Miller, "Tutorial : Program Testing Techniques", at COMPSAC '77 IEEE Computer Society, 1977.
10. P. G. Frankl, E. J. Weyuker, "An applicable family of data flow testing criteria", IEEE Trans. Software Engineering, 14(10), pp1483-1498, Oct. 1988.
11. S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information", IEEE Transactions on Software Engineering, vol. SE-11, no. 4, pp.367-375, April, 1985.
12. Y. Oh, J. Yoo, S. Cha, H.S. Son, "Software Safety Analysis of Function Block Diagrams using Fault Trees", Reliability Engineering and System Safety, Vol.88, No.3, pp215-228, 2005.

Comparing Software Measures with Fault Counts Derived from Unit-Testing of Safety-Critical Software

Wolfgang Herzner, Stephan Ramberger, Thomas Länger, Christian Reumann, Thomas Gruber, and Christian Sejkora

Division Information Technologies, ARC Seibersdorf Research,
Tech Gate Vienna, Donau-City-Straße 1,
A-1220 Vienna, Austria
{wolfgang.herzner, stephan.ramberger, thomas.laenger, christian.reumann, thomas.gruber, christian.sejkora}@arcs.ac.at

Abstract. Systematic validation and verification of safety-critical software is of crucial importance. A key precaution is intensive testing at several levels, from the entire system down to individual functional elements, the latter often carried out as unit testing. This paper presents results from a unit test performed on a C++ package from a testbed of a safety critical application at the ARC Seibersdorf research lab. After outlining the test environment and relevant characteristics of the tested software package, a detailed analysis of the test results is given. This analysis comprises fault categorization, fault distribution, relations between software metrics (like McCabe's cyclomatic complexity or the risk categories of NASA SATC), software faults, and testing efforts, and yields clues about the significance of these measures for fault probabilities. A summary of the findings and related work conclude the paper.

1 Introduction

Computer systems increasingly permeate our environment and become an indispensable part of our everyday life. Behind their most visible representatives – the PC, the mobile phone, customer electronics and related gadgets, as well as the Internet and mobile telecommunication networks – a vastly growing realm of extensively unnoticed electronic devices fly our airplanes, guide surgical operations, or control the brakes in our cars. They all rely on software, making the high safety requirements imposed by the authorities in areas where failures imply threats for human life, as for instance in the aerospace or rail domains, almost obvious. But even if 'only' an irrecoverable material loss in case of a system failure may be incurred, like in space technologies or industrial control, an extremely low failure probability is demanded. Accordingly, from the system level down to subsystems and components, quality management and software development process technologies have to be obeyed thoroughly, and especially systematic testing deserves a maximum of attention.

A well established means for fault detection is unit testing, where individual software units (functions, procedures, methods) are exhaustively tested against their requirements or specifications, under consideration of a complete as possible code or even higher coverage like path coverage. Although unit testing repeatedly is criticized due to the rather big effort it requires, as well as the fact that it does not cover integration aspects, it is also regarded as the most effective means to test individual software components for boundary value behavior and ensure that all code has been exercised adequately [4].

(We used the term 'fault' rather than 'error' in accordance with the terminology common for safety-critical systems, where faults are defined as the sources of errors. However, when discussing related work, we took over their terminology rather than impress ours.)

This paper describes the results of a unit test carried out in the accredited software test laboratory of ARC Seibersdorf research as part of overall software product assurance measures within a large European space project. The SUT (software under test) consisted of a set of C++ classes, and the task was to verify the documentation and the code based on the European standard ISO/IEC 61508 [8]. The goal of the software test was 100 % statement coverage with an appropriate set of test cases. The faults encountered during the test were classified and extensively documented.

These documented details are – of course – confidential. But we also applied a number of software measures to the SUT in order to investigate to what extent the quality metrics determined by these measures correlate with the fault distributions we found. And there are the results of these analyses which are at the focus of this paper. On class level, fault counts have been put in relation to lines of code and cyclomatic complexity [12], based on a mean fault rate per method of a class. On method level, we first give an analysis of methods with and without faults in relation to these measures, considering the risk areas defined by the NASA SATC (Software Assurance Test Center) [17], [18], as well as the effort (time spent) to carry out the tests. To some degree, the latter could be regarded as a measure for the 'readability' or 'understandability' of software, and its relation to detected faults was also of interest.

Therefore, this paper is structured as follows. In chapter 2 the test setup is described, including a quantitative description of the SUT as well as the test environment. Chapter 3 summarizes the test results and gives the chosen fault classification, while chapter 4 comprises the analysis of the relation between several software measures and the test results. Chapter 5 shortly addresses related work, while chapter 6 concludes the paper giving a summary of the findings.

2 Test Setup

The SUT consisted of more than 580 different methods in about 50 classes with an average of 12 methods per class. The total number of source code lines with statements, so excluding blank and commentary lines, was more than 16,000. Therefore each method consisted of 28 statement lines on the average.

As test tool, Cantata++ Version 2.2 of IPL was used [10], running on an Alpha station with a Tru64 UNIX© operating system. Although the source code was written in C++, the performed software tests did not focus on object oriented characteristics but were white box unit tests on a method level with the goal of 100% statement coverage.

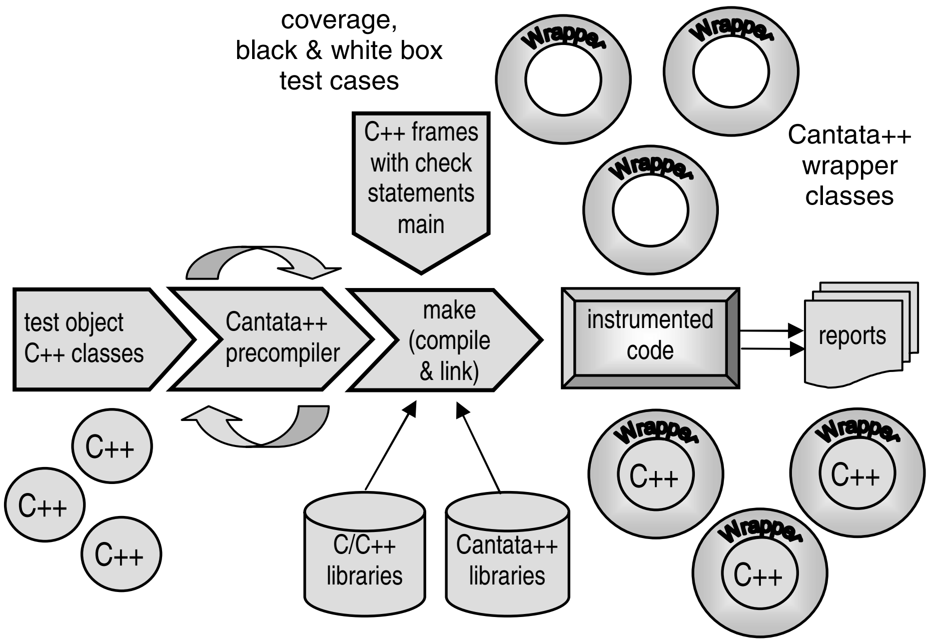


Fig. 1. Unit Test with IPL Cantata++

Figure 1 depicts the way Cantata++ handles a software unit test. Two different files have to be created by the tester in order to get an instrumented code: One that creates an instance of the class under test and another one that defines the different *wrapper* functions. Wrappers are used to modify the environment, they replace current functions by the system functions or other methods under test (MUT). The test file contains valid C++ code creating an instance of the class to call the MUT with different parameters and wrapper configurations to traverse different paths in the software. After compiling all those files with Cantata++ an executable will be created that contains the instrumented code, whose output is a text file, the *report*, listing all important test results.

The overall time it took to perform all tests on the SUT was about 40.000 minutes with an average of 67 min per method. In this context "time" considers the following activities:

- Design and implementation of test cases
- Test execution and refinement of test cases to meet the acceptance criteria (e.g. coverage)
- Configuration Management of several hundred items
- Documentation (e.g. justifications in case of non-fulfillment of acceptance criteria)
- Software Problem Reporting using WWQM
- Performing quality measures (e.g. check test cases according to checklists)

Less than 2.3% of the statements could not be reached due to defensive programming, a fact that had to be reported since it is a deviation from the demanded 100% statement coverage.

For each encountered fault in the software, we filed a *software problem report* (SPR) using the web based tool WWQM (*World Wide Quality Management*), an in-house development of ARC Seibersdorf research [13], [14], [19], providing workflow management for system (in particular, software) maintenance. This was very helpful for analyzing the results in detail and finding interesting facts about the correlation between static measures of the particular software and the observed fault rate.

3 Fault Classification

The tests as described before revealed more than one hundred faults (or SPRs, respectively), which can be categorized as documentation faults (49%), incomplete coverage (17%), coding (33%) and other faults (1%) [16]. That means, only a third of them had to be classified as coding faults, while half of all faults were classified as documentation faults.

Since we performed testing at unit level, the specification of the MUT was provided in the form of a functional description of the MUT. This covered the:

- Expected output parameters
- Expected return value
- Expected exceptions

Therefore, testing revealed either real coding faults or documentation faults as described below. Additionally, an acceptance criterion of 100% statement coverage was given which was not always achieved as mentioned below.

Coding Faults

This category contains 'real' coding faults, which might cause fatal failures during system operation.

Implementation dependencies (38%). These are semantic or implementation specific faults, like variables that will not be initialized in every case.

Wrong pointer handling (23%). Typical C or C++ faults like NULL pointer dereferencing or invalid pointer assignment.

Inversion of 'true' and 'false' (13%). In an own structure, 0 has been used to encode 'true', while values greater 0 encoded various errors. However, for comparison, the Boolean constants 'true' and 'false' have been used.

Other (26%). Various faults could not be assigned to one of the first three groups but occurred too scarcely to be split into distinct groups.

Documentation Faults

In most cases, documentation faults refer to incomplete descriptions of methods, e.g. in header files. Since, however, this may lead to erroneous usage and testing of the methods, it has been considered as fault. In addition, the documentation served as base for the test case generation. In detail, classification faults have been substructured as follows:

Value never returned (54%). A documented valid return value is never returned by the method.
Value not documented (31%). A value returned by the method has not been defined as a valid return value.
Documentation fault (10%). This relates to textual faults like quoting wrong OUT values or describing used structures incorrectly.
Doc. not sufficient (5%). The provided information is incomplete or does not specify the method properly.

Incomplete Coverage Faults

This category comprises all faults resulting from the fact that certain code lines could not be executed. In most cases, this was caused by 'defensive programming', e.g. an otherwise-branch in a switch-statement for a variable of an enumerative type, with all possible values captured by case-branches.

Defensive programming (65%). Assertions which are always true due to prior checks.
Default or else branch (35%). The use of default or else branches that cannot be reached due to checking every possible value before.

4 Fault Distribution Analysis

In this chapter the found faults are brought into relation with several measures in order to estimate their expressivity with respect to fault density prediction. However, documentation faults will not be considered, because it can hardly be argued that the description of a method in e.g. its header file is a proof for the defectiveness of its code. Also, the subcategories as presented in chapter 3 are not distinguished any further, because some of them contained too few faults for being statistically relevant.

4.1 At Method Level

A common approach of unit test analysis is to look directly at the method level, because in general methods are the primary test objects. Therefore, several method based analyses are presented here.

4.1.1 Risk Analysis

'Lines of Code (LOC)' is a simple measure which just counts the number of code lines in a source code unit. We considered only the 'net' number of code lines, i.e. no blank lines or pure commentary lines, for measuring LOC. Figure 2 shows the relationship of LOC and number of SPR per tested method. Since a unit test stops as soon as a fault is detected, SPR values higher than 1 result from retesting of faulty methods, when another fault has been detected. Not more than two retests have been necessary, which to a certain degree was caused by the fact that code has already been tested by the developers before being provided to us for unit testing.

Although figure 2 shows that no method with a LOC value higher than 200 was free of faults, it although illustrates that LOC is not a really good indicator for fault risk, because the distribution of methods with one and two detected faults is pretty similar to that of methods without detected faults. This observation coincides with results from previous works, see chapter 5.

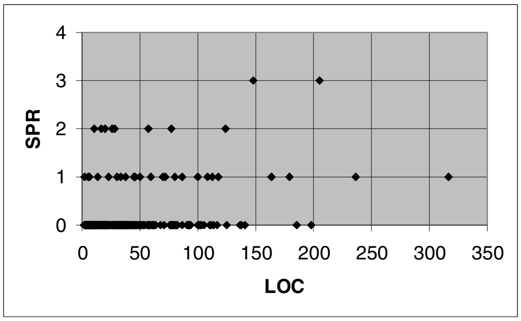


Fig. 2. Methods plotted with respect to their LOC and number of SPR

Therefore, we looked at other measures like the 'Cyclomatic Complexity (CC)', which is the number of linear independent paths as a specialization of McCabe's measure (edges – nodes + (2 * connected regions)) [12] with only one connected region. Based on LOC and CC, we derived an interesting metric developed by the NASA SATC (Software Assurance Test Center) [17], [18], that assigns every single method to one of seven 'Risk Areas' based on the lines of code and the cyclomatic complexity measures. The SATC claims that the lower the number of the risk area a method belongs to due its LOC and CC measures, the lower the probability of a potential fault in that method. Figure 3 shows these risk areas, together with the distribution of the methods tested.

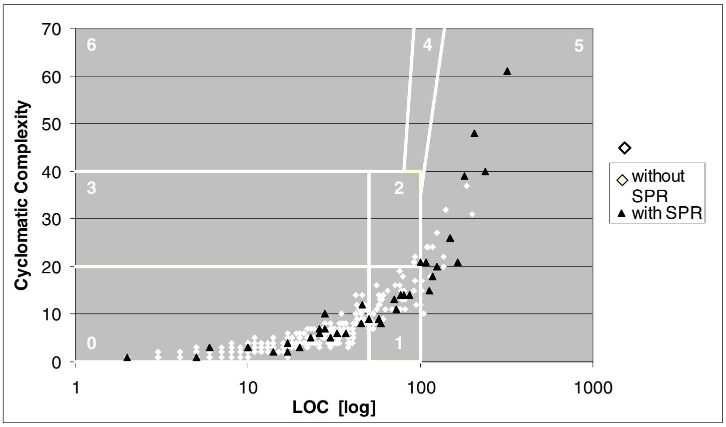


Fig. 3. Methods plotted with respect to their LOC and CC values.
Numbers and lines within the diagram field denote the SATC risk areas.

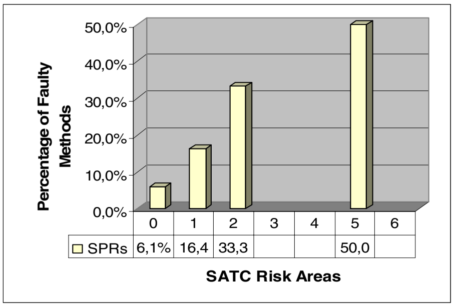


Fig. 4. Percentage of faulty methods of all per SATC risk area

What strikes one's mind is the narrow band (which would be rather straight on a linear LOC-scale) where the methods are located. But what also surprises is the apparently even distribution of faulty methods over the whole band, which seems to indicate that fault density is not correlated with risk areas. However, if the number of faulty methods (with SPR) relative to non-faulty ones (i.e. without SPR) is considered, this correlation is clearly present, as indicated in figure 4.

4.1.2 Test Effort Analysis

Another interesting aspect addresses how strong the time effort for testing each method correlates with its LOC and CC, as shown in figure 5.

In general, the time effort for testing seems to depend fairly linearly on both LOC and CC, with a – not surprising – higher amount of effort for methods with SPRs. For methods with low complexity, i.e. below 10 or 15, the testing effort was almost constant, namely approximately 50 minutes if no fault has been detected, and approximately 100 minutes if some fault has been detected.

It should be noted, that also methods with a minimal CC-value of 1 took time to be tested, namely up to three hours, as indicated in the diagram on the right side of figure 5.

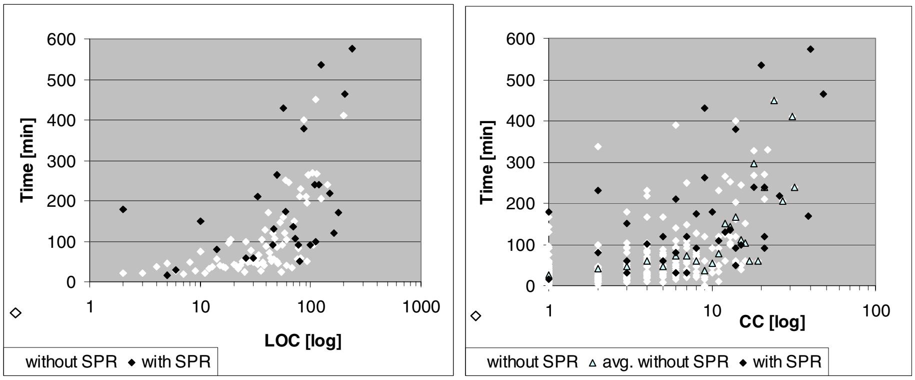


Fig. 5. Size (LOC) and complexity (CC) versus testing effort

4.2 At Class Level

Besides looking at correlations between fault distributions, LOC, CC, and test time effort as discussed before, we also wanted to examine whether these correlations remain significant at class level as well.

To enable comparison of classes based on the fault counts of their methods, a certain ‘normalization’ of these counts is necessary, in order to compensate the different numbers of methods per class. This *class fault rate* is computed by $CFr_k = SPR_k / M_k$, with k as the class index, CFr_k the class fault rate of class k, SPR_k the sum of all faults found for its methods, and M_k the number of its methods.

We further categorized classes into eight categories according to their *CFr* as shown in table 1.

Table 1. CFr categories and characteristic data

Category	CFr-Range	# Classes
0	0.0	19
1	(0.0, 0.1]	9
2	(0.1, 0.2]	11
3	(0.2, 0.3]	4
4	(0.3, 0.4]	2
5	(0.4, 0.5]	0
6	(0.5, 0.6]	0
7	(0.6, 0.7]	1

4.2.1 CFr versus LOC

When placing all classes, represented by their CFr, relative to their LOC, a distribution as shown in figure 6, results.

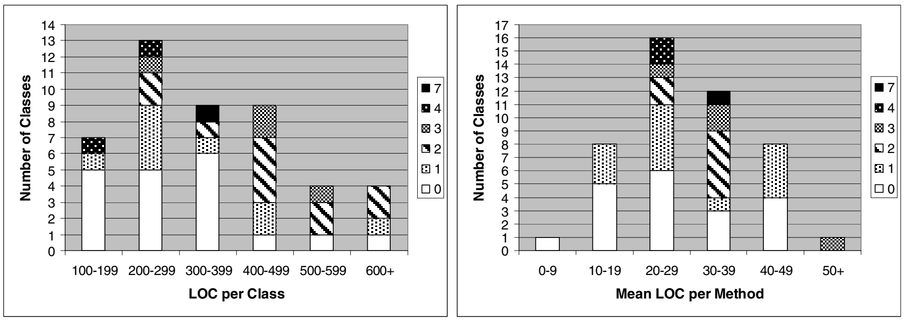
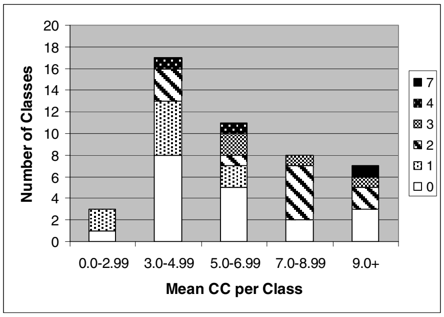


Fig. 6. Number of classes per LOC, presented by their CFr categories (0..7). Left: LOC includes all code lines of a class. Right: class LOCs divided by number of methods in class. CFr categories 5 and 6 are not presented, because no classes are contained in these categories.

Remarkably, when complete classes are considered (left diagram in figure 6), those with highest CFr have the smallest LOC-values, while in the group with largest LOCs, only classes with low CFr are present. When LOC-values are normalized with respect to number of methods per class (right diagram in figure 6), a slightly stronger correlation between LOC and CFr results, but still classes with high CFr have methods of rather small means size.

4.2.2 CFr versus CC

Figure 7 shows the relation between the mean CC per class and its CFr. It is claimed that the higher the CC, the higher the fault probability, with values above 40 best to avoid. Values up to 20 are considered ‘safe’ in general.



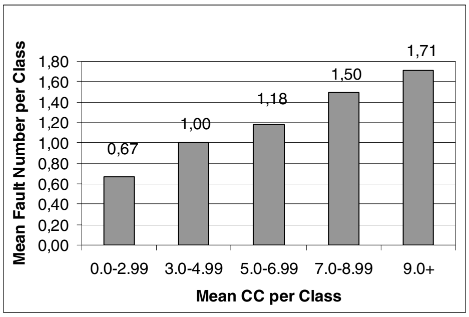
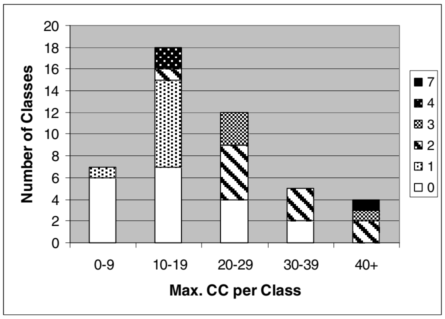


Fig. 7. Left: number of classes ordered per mean CC, presented by their CFr categories (0..7). Right: accumulated fault numbers, divided by number of classes. CFr categories 5 and 6 are not presented, because no classes are contained in these categories.

Of course, since mean CC values over all methods of a class have been taken, the abscissa range in figure 7 is smaller, since in general the majority of methods of a class have rather small CC values. Nevertheless, figure 7 shows a notably closer correlation between CFr and CC than can be found in figure 6 with respect to LOC. To the left, although classes with CFr = 0 can be found in all mean CC categories, those with higher CFr values clearly tend to lie in higher CC categories. If per CC category the number of faults of those classes contained in the respective category is divided by the number of classes in that category, an almost stunning linear correlation between CC and fault probability emerges, as shown in the right diagram of figure 7.

If mean CC values are replaced by a simpler measure, namely the maximum CC over all methods per class, a similar though less significant distribution can be found, as shown in figure 8. In this case, classes with CFr = 0 are not found with high CC values (over 40), while below 20 only two with a remarkable CFr category are presented. Both are, however, those with the second highest CFr value. But if we condense these values as before to mean fault rates per class as displayed on the right side of figure 8, the correlation between CC and fault probability on class level shows up clearly, although not as ‘linear’ as with mean CC values.



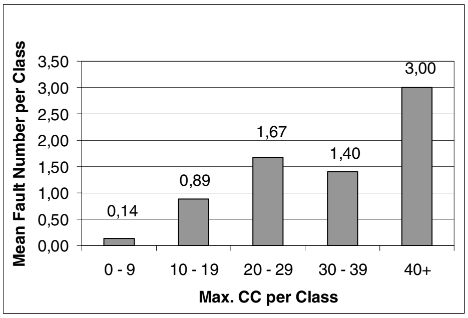


Fig. 8. Left: number of classes ordered per maximum CC over their methods, presented by their CFr categories (0..7) Right: accumulated fault numbers, divided by number of classes. CFr categories 5 and 6 are not presented, because no classes are contained in these categories

These results are interesting because maximum CC is a measure simpler than mean CC, but of comparable meaningfulness.

5 Related Work

In [4], it is stated that in contrast to a comprehensive body of theoretical work on software testing methodology there are not many published results of empirical studies in real-world software projects available, because 'Industrial staff rarely have the time to analyze past projects before being moved to other projects, and academics very rarely have access to statistically valid collections of data'. We largely share this impression, what has also been a driving force behind this paper.

Not too surprisingly, the available studies vary significantly in their general approach and methodology, as well as in scope and abstraction of hypotheses to be supported or rejected by statistical evidence, often based on fairly small samples. Other differences rendering comparison more difficult come from the tested software systems themselves, specifically from the unclear contribution of heterogeneous software engineering techniques applied for development, from the use of different implementation languages, and from the undefined coding maturity of software implementers from various academic institutions and industrial enterprises. But there are a couple of papers with a sufficient thematic overlap with our work, which shall be outlined here.

In [5] eight hypotheses, partially subdivided into sub-hypotheses, were tested against in two releases of a major commercial system. Concentrating on differences between pre- and post-release fault densities, the authors found support for theses like 'a small number of modules contain most of the faults discovered during pre-release testing, as well as operational faults (the Pareto principle)' or 'fault densities at corresponding phases of testing and operation remain roughly constant between major releases of a software system', while they found rather no support for theses like 'higher incidence of faults in all pre-release testing implies higher incidence of faults in post-release operation' or 'size metrics (such as LOC) are good predictors of number of pre-release faults in a module', which for LOC on class-level is in alignment with our observations. In contrast, we found that CC proves to be a simple and suitable predictor for fault probability, which again complies with findings previously published in [15].

Based on observation on three related software projects from the embedded systems domain, Ellims et al. found out that beside detailed design (and review), unit testing contributed most to the detection of faults [4].

[2] is, though published more than twenty years ago, still of interest. The authors analyzed the distribution of errors in modules of a medium-scale software project with respect to environmental factors like complexity, developer's experience, and reuse. Of greatest interest are the differences found between the prevalent errors in modified and in new modules. At the first glance, modifying existing modules seemed to reduce development costs but modified modules turned out to be more susceptible to errors due to misunderstanding of specifications. After thorough analysis the authors could show that there are 'hidden costs' due to the increased necessary effort for correcting the specific faults of modified modules. Another result was that module size did not account for error-proneness. The larger the modules, the less error-prone they were, even if they were more complex. This fact can be substantiated with the results of our study where modules with lower LOC count tended to exhibit a greater probability for errors.

[1], another standard in the field of empirical software quality data, investigates the distribution of error rates for design errors in product code. Based on his data on the (great) mean time to discovery, the author doubted that all design errors could be removed through testing. He assumed that any software product exhibits similar regularities in the rate behavior, irrespective of the use of the product. Consequently, his findings are appropriate for the estimation and planning of service effort a software product will need after deployment.

[11] analyzes the defect data from several wide-distribution commercial software releases in order to address the problem of quantifying the software products' field quality. The authors found out that the estimated number of defects remaining in the code constitute a metric for the field quality, rather than the estimated reliability of the product. The apparent number of defects is strongly related to the number of users of that software, with new users having a greater probability to find new defects. Additionally, it became apparent that new releases of a software stimulate the discovery of latent defects already present in the preceding release.

[3] is an analysis of typical issues related to empirical studies in the field of software testing techniques. The authors address the topics: Fault seeding, academic v/s industrial settings, need for replication of studies across different settings, and the implications of human factors in the production of error seeded test sets. They propose topics for future research, like the establishment of a standardized benchmark for testing techniques.

Common topics most authors want to shed light upon are relations between design and code complexity measures, testing methods and testing intensity (coverage), and a probability interval for the absolute number of faults in dedicated portions of the software under test. Consequently, faults should be identifiable at an early development stage, at significantly lower cost. [5] states that "(...) the various empirical studies have thrown up results which are counter-intuitive to the very basic and popular software engineering beliefs", and follows that this should be a warning to the software engineering research community.

Examples for such results are: Simple complexity measures seem to be as useful as more complicated measures ([15]). Modules where more defects are found by pre-release test coincide with modules where customers find more defects ([6]). Larger components are proportionally more reliable than smaller components ([7], cited in [5]). The concerted establishment of a wide collection of empirical data would provide the foundation for a proper evaluation of different software engineering and testing methods. We hope to be able to contribute a small part to this basis with the data and analysis presented in this paper.

6 Conclusion

Primarily, our observations confirm CC as a good measure for fault probability: Although the absolute density of faulty methods is rather constant over the CC-range of approximately 5 to 50 (see figure 3), its density relative to methods without detected faults rises significantly with increasing CC (see figures 7 and 8).

Since in our test set the ratio LOC/CC is almost constantly 6 (see figure 3), at method level LOC can to some extent be seen as an indicator for fault probability, although less significant than CC due to its higher variability (see also figure 2). However, this resemblance breaks down at class level; while CC turns out to be a remarkable measure for fault probability at class level (figures 7 and 8), this does not apply for LOC (figure 6). Perhaps, the strong, almost linear correlation between mean CC per class and mean fault number per CC class as shown at the right side of figure 7 may turn out as one of the most surprising results of this study.

The risk areas concept of SATC proves to be reasonable and valid (see figure 3), in particular, when fault probability is computed as ratio between non-faulty and faulty methods (figure 4).

Furthermore, comparison of effort with LOC indicates that from a threshold about 40 or 50, the time spent for dealing with a method increases somewhat over-linearly with LOC. Small CC values (below approximately 10) appear to have no visible effect on effort, while higher values do have, of course (figure 5).

Finally, we want to point out that within the Integrated Project DECOS (project nr. IST-511764) “Dependable Embedded Components and Systems” in the sixth European Framework (www.decos.at), which is coordinated by ARC Seibersdorf research, we are designing and implementing a Test Bench Framework, which includes unit testing besides a set of other methods and tools to fulfill validation and verification of distributed time-triggered, safety critical systems from the application system model to deployment. We expect further experimental results for a variety of V&V methods and metrics.

References

1. Adams, E. N.: Optimising Preventive Service of Software Products. IBM J. Res. Develop., Vol. 28, No. 1 (Jan. 1984)
2. Basili, V. R., Perricone, B. T.: Software Errors and Complexity: An Empirical Investigation. Communications of the ACM, Vol. 27, No. 1 (Jan. 1984)

3. Briand, L. Labiche, Y.: Empirical Studies of Software Testing Techniques: Challenges, Practical Strategies, and Future Research. Workshop on Empirical Research in Software Testing (WERST'2004). Online [PDF Document. http://www.sce.carleton.ca/squall/WERST2004/accepted_papers/LB-YL.pdf (16/02/2005)
4. Ellims, M., Bridges, J., Ince, D. C.: Unit Testing in Practice. In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), 1071-9458/04 (2004).
5. Fenton, N. E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. In: IEEE Transactions on Software Engineering, Vol. 26, Nr. 8 (2000) 797-814
6. Gittens, M. Lutfiyya, H. Bauer, M. Godwin, D., Yong W. K., Gupta, P.: An Empirical Evaluation of System and Regression Testing. University of Western Ontario and IBM. Online (PDF Document) http://www.cleanscape.net/docs_lib/ibm_cas02.pdf (2/16/2005)
7. Hatton, L.: Software Failures: Follies and Fallacies. In: IEE Review, Vol. 43, No. 2 (Mar. 1997), 49-52
8. IEC 61508, Functional Safety of Electric/Electronic/Programmable Electronic Systems, Part 1 – Part 7 (1998 – 2001)
9. IEEE 982.2-1988 "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software". IEEE Press (June 1989)
10. IPL, http://www.ipl.com/products/tools/pt400.uk.php. This web-page also provides links to technical papers.
11. Kenney, G. Q., Vouk, M. A.: Measuring the Field Quality of Wide-Distribution Commercial Software. In: Proc. of 3rd Int. Symposium on Software Reliability Engineering, Research Triangle Park, NC, USA, (1992) pp. 351-357. Online (PDF-Document) http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/Kenney/Kenney_ISSRE92.pdf (2/16/2005)
12. McCabe T. J.: Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. National Bureau of Standards, Washington (1982)
13. Moschitz, M., Thuswald, M., Weber, E.: WWQM2 im Jahr 2000 (in German only). Internal report, ARC Seibersdorf research (2001)
14. Moschitz, M., Studer, M., Fuhrmann, S., Weber, E., Zoffmann, G.: WWQM im Jahr 2002 (in German only). Internal report, ARC Seibersdorf research (2003)
15. Ohlsson, N., Alberg, H.: Predicting Fault-Prone Software Modules in Telefone Switches. In: IEEE Transactions on Software Engineering, Vol. 22, No. 12 (Dec. 1996)
16. Ramberger, S., Gruber, T., Herzner, W.: Experience Report: Fault Distribution in Safety-Critical Software and Software Risk Analysis Based on Unit Tests. In: Dadam, S., Reichert, M. (Hrsg.), INFORMATIK 2004, Band 1, vol.P-50 of Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI). 2004.
17. Rosenberg L.H.: Applying and Interpreting Object Oriented Metrics. Software Technology Conference, April 1998 (http://satc.gsfc.nasa.gov/metrics as of Feb.2003)
18. Software Assurance Technology Center. http://satc.gsfc.nasa.gov as of Feb.2003
19. Public WWQM test version at http:://www.wwqm.at, with "guest" account (no password)

Automatic Analysis of a Safety Critical Tele Control System*

Edoardo Campagnano[1], Ester Ciancamerla[1], Michele Minichino[1], and Enrico Tronci[2]

[1] ENEA CR Casaccia, Via Anguillarese, 301, S. Maria di Galeria, 00060, Roma, Italy
`{ciancamerlae, minichino, edoardo.campagnano}@casaccia.enea.it`

[2] Dipartimento di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy
`tronci@di.uniroma1.it`

Abstract. We show how the Murφ *model checker* can be used to automatically carry out safety analysis of a quite complex hybrid system tele-controlling vehicles traffic inside a safety critical transport infrastructure such as a long bridge or a tunnel. We present the Murφ model we developed towards this end as well as the experimental results we obtained by running the Murφ verifier on our model.

Our experimental results show that the approach presented here can be used to verify safety of critical dimensioning parameters (e.g. bandwidth) of the telecommunication network embedded in a safety critical system.

1 Introduction

Because of technological as well as economical reasons, the number of systems relying on wireless telecommunication (telco) networks is always increasing. This is also happening for *safety critical systems*. This poses new challenges to the safety analysis work. In fact, the telco network behaviour needs to be modeled in a fairly accurate way in order to formalize the relationship between telco network parameters (e.g. *bandwidth*) and the system safety property being investigated.

We show how the above is possible by presenting a case study on the analysis of a safety property for a *Tele Control System* (TCS), developed in the frame of the European project *SAFETUNNEL* [11].

The goal of TCS is to take active measures to improve safety in the *Critical Transport Infrastructure* (CTI) it controls, namely a tunnel. More specifically, TCS aims at reducing the number of accidents inside alpine road tunnels, exploiting GPRS (*General Packet Radio Service)*) communication between instrumented vehicles and a *Tunnel Control Centre* (TCC). TCS implements preventive safety functions, namely: vehicle prognostics, vehicle tunnel access control, vehicle speed and distance control, dissemination of emergency message.

* Contact Author: Michele Minichino, Tel.: +39 06 3048 3407, Fax: 06 3048 6511.

We present a model of TCS and an automatic analysis of it via model checking [12]. Our goal is to show that TCS operates in a safe way, that is no dangerous situation can arise from installation and usage of the TCS in our CTI.

More specifically, our analysis focuses on the interaction of TCS telco network dimensioning with TCS preventive safety functions. We formally check, via model checking, that the telco dimensioning, in terms of bandwidth, guarantees TCC ability to safely handle different tunnel scenarios. Namely: normal system operational mode (registrations, deregistrations, anomaly situations and emergency situations), emergency scenarios (i.e. dissemination of emergency information).

Basically, our present work is about TCS validation by modeling along the lines of [1]. In fact, in our case, only a limited number of field tests can be run on the actual system. This is because measures requiring long observation times inside the infrastructure (that has to be closed to the ordinary vehicular traffic, with loss of availability and money) should be kept to a minimum. Moreover measures which would require irreproducible infrastructure scenarios (i.e occurrence of incidents and emergency scenarios) cannot simply be done. From the above considerations stem the importance safety and performance analysis on the system model.

TCS is a quite large *hybrid system*, that is a system with continuous as well as discrete state variables. Automatic analysis of *Hybrid Systems* poses formidable challenges both from a modeling as well as from a verification point of view. In fact the simultaneous presence of continuous and discrete variables may lead very quickly to *state explosion*, thus preventing completion of the verification process.

Many verification tools (*model checkers*) are available for automatic verification of hybrid systems. Examples are: HyTech [9,3,2] and UPPAAL [10,18]. Also tools originally designed for hardware verification have been used for hybrid systems verification. E.g. in [17] SMV [12,16] has been used for verification of chemical processing systems.

In this case study we use the CMurφ [5,4] verifier since both HyTech and SMV could not complete the verification task because of state explosion. This is in agreement with our previous experience in hybrid systems verification [14].

CMurφ is the Murφ verifier [6,13] extended with (finite precision) real numbers [14], caching and disk based algorithms [15,5].

Automatic timeliness verification with the Murφ verifier and performability analysis of TCS telco network has also been studied, respectively, in [8], [7].

Our main contributions here can be summarized as follows. We sketch TCS features (Section 3), present our modeling of the TCS system (Sections 4, 4.1, 4.2, 4.3, 4.4), present a formalization of the main TCS safety requirement (Section 5) and finally give experimental results showing effectiveness of our approach (Section 6). Lack of space prevents us from giving the Murφ model of TCS.

2 Basic Notions

A *Finite State System* (FSS) $\mathcal{S}$ is a 4-tuple (S, I, A, R) where: S is a finite set (of states), $I \subseteq S$ is the set of initial states, A is a finite set (of *transition labels* or *events* or *actions*) and R is a relation on $S \times A \times S$. R is usually called the *transition relation* of $\mathcal{S}$. We define the set $\texttt{next}(s)$ of successors of state s as follows: $\texttt{next}(s) = \{s' | \exists a R(s, a, s')\}$.

The set of *reachable states* of $\mathcal{S}$ (notation **Reach**($\mathcal{S}$)) is the set of states of $\mathcal{S}$ reachable in zero or more steps from I.

A *trace* π of $\mathcal{S}$ is a finite or infinite sequence $\pi \equiv s_0, a_0, s_1, a_1, \ldots$ s.t.: $s_0 \in I$ and for $i = 0, 1, \ldots$ $R(s_i, a_i, s_{i+1})$ holds. We also write $\pi(i)$ for $s(i)$.

In the following we will always refer to a given (once and for all) system $\mathcal{S}$ = (S, I, A, R). Thus, e.g., we will write **Reach** for **Reach**($\mathcal{S}$). Also we may speak about the set of initial states I as well as about the transition relation R without explicitly mentioning $\mathcal{S}$.

Let $\mathcal{B} = \{0, 1\}$ the set of boolean values. An *invariant* for $\mathcal{S}$ = (S, I, A, R) is a map φ from S to $\mathcal{B}$. We say that $\mathcal{S}$ satisfies invariant φ iff for all $s \in$ **Reach** $\varphi(s) = 1$. That is, if for all reachable states of $\mathcal{S}$, φ holds.

Safety properties are modeled using invariants. That is, an *error state* or an *undesired state* is a state that does not satisfy the given invariant.

Basically, using a suitable high level language, a *model checker* takes as input the definitions of an FSS $\mathcal{S}$ and of an invariant φ for $\mathcal{S}$ an returns `PASS` if $\mathcal{S}$ satisfies φ, `FAIL` otherwise. Moreover, when a model checker returns `FAIL`, it also returns a finite trace $\pi \equiv s_0, a_0, s_1, a_1, \ldots s_k$, of $\mathcal{S}$ leading to an error state, that is we have $\varphi(\pi(k)) = \varphi(s_k) = 0$.

From the above follows that, *given* a system $\mathcal{S}$ and an invariant φ, a model checker *automatically* carries out a a *reachability analysis*, i.e. the computation of all reachable states, for $\mathcal{S}$, looking for undesired states (i.e. states not satisfying invariant φ).

We plan to use CMurφ [5,4] extended with real numbers [14] to analyze hybrid systems. For this reason we model hybrid systems as *Discrete Time Systems* (DTSs). We show the easy relationship between DTSs and FSSs using a toy example. Let us consider the DTS $\mathbf{x}$ defined by Equation 1, where $\mathbf{x}(t)$ is the state value at time t and $\mathbf{d}(t)$ is the disturbance value at time t.

$$\mathbf{x}(t+1) = \begin{cases} \mathbf{x}(t) + \mathbf{d}(t) \text{ if } \mathbf{x}(t) \leq 3 \\ \mathbf{x}(t) - \mathbf{d}(t) \text{ otherwise} \end{cases} \qquad \forall t[\mathbf{d}(t) \in \{0, 1, 2\}], \qquad \mathbf{x}(0) = 0. \quad (1)$$

$$\varphi_1(\mathbf{v}) = (\mathbf{v} \leq 5) \qquad\qquad \varphi_2(\mathbf{v}) = (\mathbf{v} < 5) \qquad (2)$$

Fig. 1 shows the FSS corresponding to the DTS defined by Equation 1. The initial state $\mathbf{x}(0) = 0$ is shown with an arrow in Fig. 1, where nodes are labeled with state values and edges are labeled with action (disturbance, in our case) values.

Equation 2 defines possible invariants for system $\mathbf{x}$ in Equation 1. A model checker taking as input the pair ($\mathbf{x}$, φ_1) will return `PASS` since all reachable states of $\mathbf{x}$ are less than or equal to 5. On the other hand a model checker with input ($\mathbf{x}$, φ_2) will return `FAIL` with the following trace (counterexample) $0, 1, 1, 2, 3, 2, 5$.

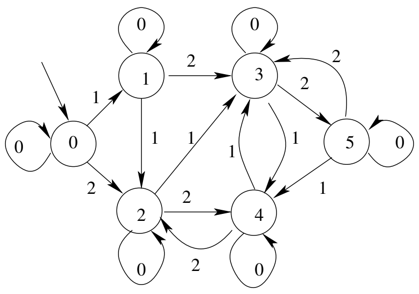


Fig. 1. FSS for the discrete time system in Equation 1

3 System Overview

In this Section we give a high level description of our TCS architecture. The remaining Sections will gradually zoom in TCS components showing how our Murφ model is organized.

The goal of TCS is to monitor and control vehicle (mainly trucks) traffic inside the CTI area. This is done by equipping each vehicle with suitable sensors and actuators (e.g. to measure and control the distance from the preceding vehicle) and with telecommunication devices (to communicate with the control center).

TCS consists of three main subsystems: *Vehicles*, *Telecommunication network* (TLC) and, finally, the *Tele Control Center* (TCC).

The *Tele Control Center* (TCC) manages the vehicles in the CTI area. The TCC-vehicle communication protocol is defined with *Message Sequence Charts* (MSCs) which also define the telecommunication network load, since they define the number of bytes traveling in the communication channels. In case of an accident the TCC sends to all vehicles in the CTI suitable directives to escape from the accident area.

This is the most stressful situation for the telecommunication network. Since our main goal here is to verify the telecommunication network dimensioning, we will just focus on the case in which, for some reason (e.g. an accident), the TCC needs to send a given (*emergency*) message to all vehicles.

As far as we are concerned, vehicles are equipped as follows: 1) Fuel level sensors, distance sensors, oil level sensors, etc; 2) GPRS telecommunication devices; 3) Automatic Cruise Control (ACC), which takes from the TCC the max speed and min distance and actuates vehicle throttle and brakes accordingly.

A vehicle equipped with the above devices is also called a *mobile station.*

For safety reasons a vehicle must be an autonomous system, i.e. it should work safely also when the TCC or the telecommunication network are not working. This is why vehicles are equipped with an *Automatic Cruise Control* (ACC) that keeps the vehicle speed below a given threshold and the distance of the vehicle from the preceding one above a given threshold.

Communication between mobile stations (vehicles) and the TCC essentially exploits the GPRS technology used to support communication between mobile stations and TCC inside the whole CTI area.

CTI GPRS network consists of a set of *Base Stations* situated inside the CTI. Each Base Station supports traffic for a certain number of *Carriers*. The number of carriers per base station depends on the type and configuration of the base station model.

Using *Time Sharing* policies each carrier, in turn, is split into 8 *Time Slots*. This is the channel used for actual data transmission. The time slot channel has a transmission speed of 10.22 kbps. Theoretically a GPRS terminal can use up to 8 time slots in *uplink* (UL) plus 8 in *downlink* (DL). Typically, commercial terminals use 6 time slots for uplink and downlink.

As an example, assuming we have a 3 carriers base station, we have available $3 * 8 = 24$ time slots for each installation.

The following alternative working hypothesis have been considered in the GPRS dimensioning: 1) The max *bit rate* (UL + DL) for each mobile station is 5 kbps, thus 2 vehicles can share one time slot; 2) The max *bit rate* (UL + DL) for each mobile station is 2 kbps, thus 5 vehicles can share one time slot. Of course the first solution gives faster communication, but requires more carriers. The second solution saves on the number of carriers, yielding however slower communication.

4 TCS Model

We use the Murφ programming language to define our model and the Murφ verification engine to check that our model meets given safety requirements. Murφ uses a Pascal-like programming language to define model dynamics. This makes the definition of complex systems quite easy, since an *object oriented* modeling approach can be followed.

Because of lack of space we cannot present the actual Murφ code of our model. We will just describe the main subsystems forming our systems as well as their interactions.

Murφ constants are our TCS model parameters. Some of our constants are suggested by [11], others have been obtained from various (e.g. physical) considerations.

Murφ data structures are our TCS model *objects* (e.g. vehicles, etc). Murφ functions are used to define the dynamics of our TCS model.

As usual we follow the convention of ending function names with `()`. Function names used in this section correspond exactly to those in the Murφ model.

We model TCS as a *discrete time system* with sampling time $T = 100$ms [11].

A high level view of TCS consists of three main objects (Figure 2). Namely, (an array of) mobile stations (i.e. vehicles), the *Telecommunication Network* (TLC), the *Tele Control Center* (TCC).

Figure 2 shows some of the (Murφ) functions (`SendRequest()`, `AssignChannel()`, `CheckBarrier()` and `BlueToothTrigger()`) implementing

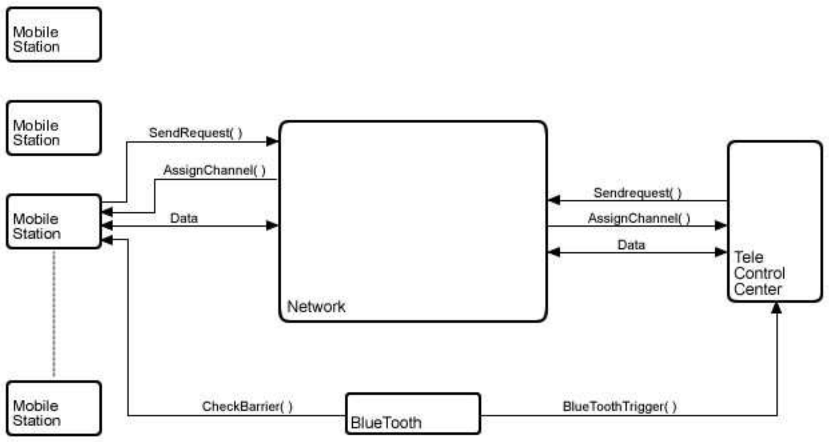


Fig. 2. Model top view

the interaction between (top) TCS objects, namely *Mobile Stations*, TCC and TLC.

Mobile Stations and TCC communicate via the TLC which consists of a GPRS network and a system of antennas used to check vehicle parameters (e.g. position) at the CTI barriers (`CheckBarrier()` in Figure 2) and possibly to send messages to the TCC (`BlueToothTrigger()` in Figure 2).

For GPRS communication a channel must be assigned to the peers. This is modeled as follows (Figure 2). The sender asks for a communication channel `SendRequest()` to the *Network Manager*. Once such channel is assigned to the sender (using `AssignChannel()`) the communication can take place. That is the sender can send its message to the receiver (`Data`).

Note that the CTI itself does not appear in Figure 2. This is because the *CTI* status does not change over time. Thus it can be simply modeled using its physical constants.

For example, constant `TUNNEL_LENGTH` defines the physical length of the CTI under consideration. Constant `APPROACHING_LENGTH` defines the distance outside CTI entrances that we still consider relevant for our modeling (*CTI area*). Constant `TOOTH_DISTANCE` gives the distance of the first Bluetooth barrier from the CTI entrance (of course, on both sides of the CTI). As a result, position (in meters) of the four CTI barriers can be easily computed. Thus, in our TCS model, to formalize the fact that a vehicle has passed a certain barrier it suffices to compare the vehicle position with the barrier position.

4.1 Vehicles

A single vehicle is modeled using a record (named `Vehicle`). Each record `Vehicle` field models a vehicle feature (e.g. `position`, `speed`, etc) needed in order to define the dynamics of our model. In other words, record `Vehicle` holds the vehicle state information.

Our CTI has one lane for each direction. Each lane is modeled with an `array` of size `NUMBER_OF_VEHICLES_PER_LANE` of vehicle records.

A vehicle can be a car or a truck. Each vehicle is equipped with suitable communication devices [11]. For this reason, in our context, vehicles are also called *mobile stations* or *terminals* (when dealing with TLC network issues).

When modeling a vehicle dynamics we also take into account its acceleration and deceleration characteristics.

4.2 The Tele Control Center

The TCC consists of four interacting subsystems: 1) Communication devices; 2) Constant directives (storing system parameters)p 3) Registered vehicle data (storing information about registered vehicles); 4) Right monitoring devices (handling the *tight monitoring* procedure to be describe din Section 4.4).

For example, among the TCC constants directives (parameters) we have `STANDARD_RECOMMENDED_SPEED` (70 Km/h) `STANDARD_RECOMMENDED_DISTANCE` (150 m). If an anomaly occurs in the monitored area TCC suitably recomputes these values.

For each vehicle v, TCC stores information about v as well as information about the messages exchanged between TCC and v.

Our model for the Tele Control Center consists of: 1) *CTI Status Variables*, storing all information (directives) to be sent to vehicles (e.g. *Recommended Speed* AND *Recommended Distance*); 2) The I/O system handling GPRS communication with the mobile stations; 3) Administrative information to make decisions about messages to be sent to vehicles.

4.3 The Telecommunication Network

The TLC network is one of the main target of our analysis. More specifically, our goal is to check that TLC dimensioning guarantees TCC ability to safely handle emergency situations. In fact, when an emergency occurs, TCS sets up a particular emergency procedure involving the TCC as well as many vehicles. This is the more demanding situation for the telco network.

Figure 3 shows our model for the telecommunication network. We view the telecommunication network as a set of (*virtual*) channels and a manager that handles virtual channel assignments and releases.

To save on the state space dimension, we only model the GPRS network and ignore other components.

In the GPRS architecture each base station can have up to 12 carriers, although typically a base station has 3 or 4 carriers. In our setting we can assume that each base station can have at least 6 carriers because of the high expected traffic volume. In the following we denote with C the number of carriers for each each base station.

Each carrier can have up to 8 time slots to be used for communication. However usually at most 6 are used. In the following we denote with N_{slots} the number of time slots for each carrier.

In the following we denote with T_{speed} the number of bits per second that a time slot can transmit. With the network configuration envisaged in [11] we have: T_{speed} = 10.22 kbps = 10465 bps.

The same time slot can be used by more than one terminal (vehicle). We denote with V_{ehic} the number of vehicles sharing the same Time Slot.

For example if the max bit rate per vehicle (UpLink + DownLink) is 5kbps we can allocate 2 vehicles on the same time slot.

We define as *Virtual Communication Channel* or just *channel* the transmission bandwidth *ideally* allocated to each terminal (vehicle). In the previous example we have 2 channels with a transmission speed of 5kbps for each slot.

We denote with B the number of base stations available.

Given the network technology (e.g. GPRS for us) the number of channels `NUMBER_OF_CHANNELS` and their speed `CHANNEL_CAPACITY` are project requirements for the network design. The following relations hold:

$$\texttt{NUMBER\_OF\_CHANNELS} = BCN_{slots}, \ \texttt{CHANNEL\_CAPACITY} = T_{speed}/V_{ehic}.$$

For example, with our data (T_{speed} = 10.22 kbps, $V_{ehic} = 5$) we have: `CHANNEL_CAPACITY` = T_{speed}/V_{ehic} = 10465/5 = 2093 bps.

Here we are only interested in transmission capacity. For this reason we consider channels as basic elements of our modeling.

Communication channels, of course, can be implemented with many technologies. The only difference resides in the network architecture (e.g. number of base stations, carriers, etc) needed to meet the given network specifications, `NUMBER_OF_CHANNELS`, and `CHANNEL_CAPACITY` for us.

In other words, `NUMBER_OF_CHANNELS` and `CHANNEL_CAPACITY` define the *external* view of the telecommunication network and are indeed the design parameters of the network itself. Since our goal is to study the interaction of the telecommunication network with the other TCS subsystems `NUMBER_OF_CHANNELS` and `CHANNEL_CAPACITY` are indeed a good abstraction of the network. That is, they are what the other TCS subsystems *see* of the network.

Of course the above computation of B assumes that each base station covers most of the area of our interest. This is a reasonable assumption in the case of CTI area.

Communication set up is done, once and for all, from each vehicle upon entering CTI area. This establishes a *communication link* between the terminal (vehicle) and the TCC. During this setup the vehicle sends to the TCC administrative information such as vehicle identifier, etc. When a terminal (vehicle) wants to communicate with the TCC it must look for an available channel, that is a channel not in use by another vehicle. Only once such available channel is found communication can take place. Thus each communication round is preceded by a *channel search* phase.

A terminal (vehicle) may loose its communication link with the TCC. In such cases the interaction protocol with the TCC is such that the lost link cannot be recovered. Thus if a vehicle looses its communication link, it is no more connected to the TCC.

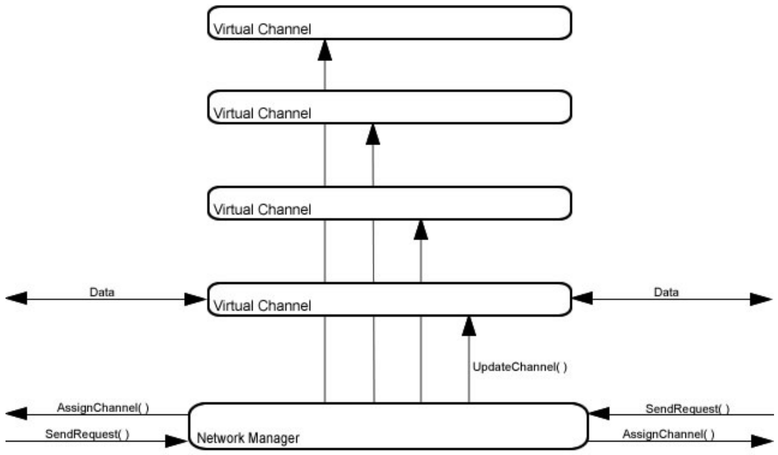


Fig. 3. Telecommunication Network

4.4 Communication Protocols

The protocols used in the TCS are defined by using *Message Sequence Charts* (MSCs). In particular we have a *Vehicle Registration Procedure* (VRP), a *Vehicle Deregistration Procedure* (VDP), a *Tele Control Application Procedure* (TAP), a *CTI Exit Procedure* (CEP), an *Emergency Procedure* (EP). To make our model working we have to model all such procedures. For space reasons, however, here we only show (Figure 4) the *Emergency Procedure* which is needed to define our safety requirement.

Many different kind of emergencies, with different severity levels, each requiring specific recovery procedures, are considered in CTI.

However, emergency ranking often requires a human intervention. This is hard (if possible at all) to model in our framework. On the other hand our goal here is to evaluate safety of the Tele Control System consisting of the TCC, the TLC network and the vehicles. For this reason we just consider the emergency situation that is more demanding for the TLC network. This happens when the TCC has to broadcast an emergency message to all vehicles in the CTI area, Figure 4.

Our goal here is to simulate an accident blocking traffic on both lanes. In this case TCC sends to all vehicles a request to stop. Thus in our model we have a procedure `SimulateAccident()` that stops (suddenly) a given vehicle at a given point in the CTI. That vehicle then sends a `DetectedAnomalyMessage` to the TCC. Such message send to the TCC the vehicle id, the kind of accident, etc.

Upon receiving the `DetectedAnomalyMessage` message the TCC, once it has determined the nature of the emergency, starts the procedure in Figure 4. More specifically, once the TCC has determined the nature of the emergency, it sends a recovery strategy using the message `ActivateRepairingPlanningRequest`

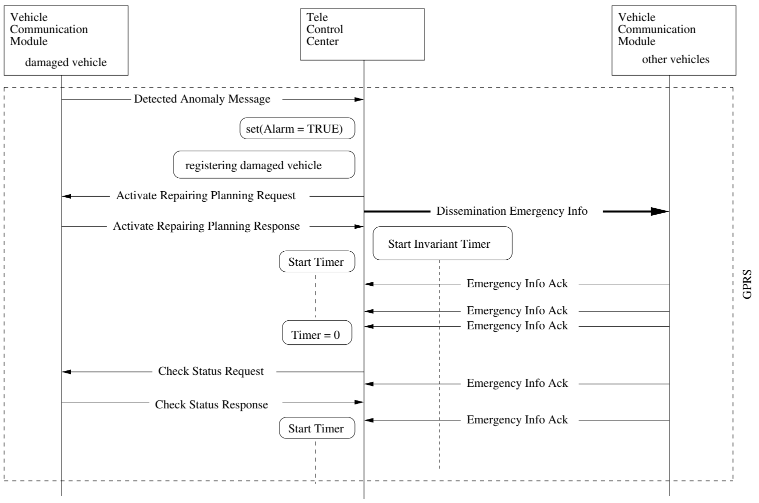


Fig. 4. Emergency Procedure

(left side of Figure 4). At the same time TCC sets to `true` the TCC alarm field and registers the vehicle involved in the emergency in order to activate a *Tight Monitoring* (TM) procedure. The TM procedure tells TCC to check the vehicles status with a higher frequency than usual. Moreover one (virtual) channel is reserved for each vehicle under tight monitoring. The mobile station (vehicle) answers the `ActivateRepairingPlanningRequest` message with a `ActivateRepairingPlanningResponse` message.

The right side of Figure 4 shows that notice of a serious emergency (incident) must be broadcasted to all vehicles in the CTI area. This, together with the ongoing TM puts a nontrivial load on the TLC network. Checking that the TLC network, under such condition, can deliver the emergency notification, to ALL vehicles in the CTI area within an assigned time constraint, is the safety requirement what we want to verify here.

Of course satisfaction of such requirement depends on the number of virtual channels available, which in turn depends on the TLC network dimensioning.

Moreover we must consider that GPRS technology, used for our TCS, does not allow *one to many* communications. Thus the broadcast needed in case of the above mentioned emergency is simulated by the TCC by sending sequentially to each vehicle in the CTI area the emergency message.

The message to be *broadcast* to all vehicles is `Dissemination_Of_Emergency_Info` and its length is 200 bytes (the longest message of all here). This message transmit an updated version of the emergency exits map, a strategy to leave the accident area as well as TCC notes (if any).

Summing up, we are going to analyze the scenario in which there is one vehicle that requires tight monitoring and TCC that broadcasts the `Dissemination_Of_Emergency_Info` message. This is the most stressful situation (assuming *single vehicle failure*) for the TLC network.

Note that, the case in which we have n channels and $(k + 1)$ vehicles requiring tight monitoring and (at the same time) broadcasting of the `Dissemination_Of_Emergency_Info` can be treated as the case in which we have $(n - k)$ channels and one vehicle requiring tight monitoring *and* broadcasting. This is because each vehicle under tight monitoring reserves a channel which is not released until the tight monitoring is over.

5 Requirements

Murφ defines requirements by using *invariants*. An invariant is a condition that all reachable states must satisfy. In other words, if a reachable state does not satisfy the given invariant we have a reachable undesired (error) state. The verifications task is to check if it is possible for the given system to reach an error state, i.e. to reach a state that does not satisfy the given invariant.

In general there are many invariants to check, one for each requirements. Here we will discuss only the main invariant for our system.

Our invariant asks that the time needed by the TCC to broadcast the `Dissemination_Of_Emergency_Info` message (Section 4.4) be below given threshold `TIME_TO_FAULT`.

The TCC, Upon receiving the `Detected_Anomaly_Message` from the vehicle:

- handles, if possible, vehicle involved in an accident;
- sends (broadcast) to all N registered vehicle a `Dissemination_Of_Emergency_Info` message;
- sets to 0 the value of our auxiliary variable `ReceivedAcks` counting the number of ack's (`Emergency_Info_Ack`) received in response to the `Dissemination_Of_Emergency_Info` message;
- initialize our timer `timer` to `TIME_TO_FAULT`.

Depending on channel availability some messages will get sent immediately, some will have to wait accordingly to the rules described in Section 4.3.

Upon receiving message `Dissemination_Of_Emergency_Info`, each mobile station will send to the TCC a `Emergency_Info_Ack` message. The TCC, in turn, increments by 1 counter `ReceivedAcks` for each `Emergency_Info_Ack` message received.

At each sampling time, variable `timer` is decremented by `SAMPLING_TIME`.

Our invariant asks that *it does not take too much to broadcast the emergency info to all vehicles in the CTI area.* That is, (`timer` $\neq 0$ or `receivedAcks` $= N$).

Using Murφ syntax this is written as follows.

```
Invariant "Too much time to deliver"
!(timer = 0.0) | receivedAcks = registeredVehicles;
```

That is, not too much time is elapsed (`!(timer = 0.0)`) or all vehicles have got the `Dissemination_Of_Emergency_Info` message (`receivedAcks =` registeredVehicles).

Of course the more virtual channel we have, the more chances we have to make our invariant true.

6 Experimental Results

In this Section we describe our verification experiments and show our experimental results.

Our invariant has been defined in Section 5.

What remains to be defined is the constant `TIME_TO_FAULT` denoting the maximum time by which all emergency messages have to be sent.

Taking `TIME_TO_FAULT` too large would make our verification uninteresting. On the other hand, taking `TIME_TO_FAULT` too small would give us *false positives* i.e. errors that indeed do not occur in the actual system.

We estimate a reasonable value for `TIME_TO_FAULT` as follows. Let v be the vehicle suggested speed inside the CTI and let d be the minimum distance among vehicles in the CTI. Suppose that a vehicle speed suddenly drops to 0 (stop). The following vehicle will bump into such stopped vehicle after

$$T_{bump} = d/v$$

Assuming (our case) that v = 70Km/h and d = 150m, we have T_{bump} = 7.7 seconds. Considering some *lead time* the above calculation suggests us to set `TIME_TO_FAULT` to 5 seconds. That is we ask that within 5 seconds all vehicles in the CTI are reached by the emergency message broadcasted by the TCC.

Two parameters can (and do) lead to state explosion: the number of vehicles and nondeterminism in the inter-arrival times between vehicles.

Thus, to avoid state explosion, we scale down our model as follows.

– We limit the number of vehicles in the tunnel area.
– We set the inter-arrival time (`ENQUEUING_TIME`) to 5 seconds for 70% of all vehicles. The remaining 30% vehicles have a non deterministic interarrival time in the interval [`ENQUEUING_TIME` - 1, `ENQUEUING_TIME` + 1].

Figure 5 shows the experimental results we obtained with Murφ.

Column *Vehicles* gives the total number n of vehicles in the CTI area (namely we have $n/2$ vehicles per lane).

Column *Channels* gives the minimum number of virtual channels needed to pass verification. For example with 10 vehicles we need at least 4 channels to satisfy our invariant. If we use 3 channels our invariant fails.

Column *Rules* gives the number of rules fired by the Murφ verifier during verification.

Column *Time* gives the time (in seconds) needed to complete our verification.

Column *Reach* gives the number of reachable states.

Column *State Size* gives the number of bit used by Murφ to represent each state.

The results in Figure 5 have been obtained using Murφ 4.2 [4] with 200 MB RAM (option `-m200`) bit compression and hash compaction enabled (options `-b -c`) on a 800 MHz Pentium 3 Linux PC. Note that computation times in Figure 5 depend on the size of the set of *reachable* states (column *Reach*). The latter, in turn, depends on *both* number of vehicles and number of channels.

Of course we may use Figure 5 to dimension our TLC network. It is interesting to compare the dimensioning obtained from Figure 5 with that obtained from the approximate worst case analysis of the TLC network.

Figure 6 plots our results from Figure 5 (bottom curve) as well as the curve obtained from the TLC network dimensioning (top curve) suggested in [11]. On the x axes we have the number of vehicles in the CTI area (column *Vehicles* of Figure 5). On the y axes we have the minimum number of virtual channels that the TLC network should have (column *Channels* of Figure 5).

The exact analysis via model checking shows (Figure 6) that we may save on the virtual channel (and thus on the TLC network size) without compromising safety. In other words, our analysis allows us to estimate the *robustness* of our dimensioning, i.e. how many channel we may lose without compromising safety.

Vehicles	Channels	Rules	Time (Sec)	Reach	State Size (bits)
10	4	77942	1798	26332	2890
20	9	58312	4960	19702	4366
30	12	48477	8443	16379	6294
40	16	98730	31041	33354	8381
50	19	79100	37915	26724	10322
60	22	59470	40916	20094	12263
70	25	170875	152150	57735	14306
80	28	58730	65170	19844	16260
90	31	54085	77480	18273	18214

Fig. 5. Murphi TCS model experimental results

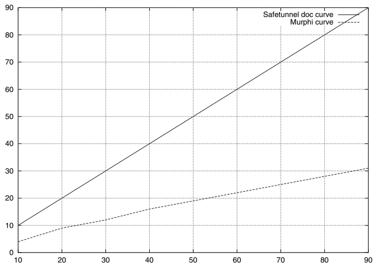


Fig. 6. Comparison between Murphi TCS model results from Figure 5 and TLC dimensioning in [11]

7 Conclusions

Our experimental results (Section 6) show that the approach presented here can be used to verify safeness of critical TLC network dimensioning parameters (namely bandwidth) as well as *robustness* w.r.t. safety of the TLC network dimensioning.

The main obstruction to be overcome is *state explosion*. Thus, in order to verify larger hybrid systems more efficient model checking algorithms are needed.

References

1. A. Bobbio, E. Ciancamerla, M.Minichino, and E. Tronci. Stochastic and functional analysis of a public mobile network in a safety critical context. In *European Safety & Reliability Conference (ESREL'05)*, June 27-30 2005.
2. A user guide to hytech: http://www.eecs.berkeley.edu/~tah/HyTech.
3. Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. Softw. Eng.*, 22(3):181–201, 1996.
4. Cached murphi web page: http://www.dsi.uniroma1.it/~tronci/cached.murphi.html.

5. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *STTT*, 6(4):320–341, 2004.
6. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525. IEEE Computer Society, 1992.
7. E. Ciancamerla and M.Minichino. Performability measures of the public mobile network of a tele control system. In *23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP'04)*, volume 3219 of *Lecture Notes in Computer Science*, pages 142–154. Springer, September 21-24 2004.
8. E. Ciancamerla, M.Minichino, S. Serro, and E. Tronci. Automatic timeliness verification of a public mobile network. In *22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP'03)*, volume 2788 of *Lecture Notes in Computer Science*, pages 35–48. Springer, September 23-26 2003.
9. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1):110–122, dec 1997.
10. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL: Status and developments. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 456–459. Springer, 1997.
11. Project IST – 1999 – 28099 SAFETUNNEL. `http://www.crfproject-eu.org`.
12. Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
13. Murphi web page: http://sprout.stanford.edu/dill/murphi.html.
14. G. Della Penna, B. Intrigila, I. Melatti, M. Minichino, E. Ciancamerla, A. Parisse, E. Tronci, and M. V. Zilli. Automatic verification of a turbogas control system with the murφ verifier. In Oded Maler and Amir Pnueli, editors, *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, April 3-5, 2003, Proceedings*, volume 2623 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2003.
15. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Integrating ram and disk based verification within the murφ verifier. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*, pages 277–282. Springer, 2003.
16. Smv web page: http://www.cs.cmu.edu/~modelcheck/.
17. A. L. Turk, S. T. Probst, and G. J. Powers. In Oded Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 1997.
18. Uppaal web page: http://www.docs.uu.se/docs/rtmv/uppaal/.

A Formal Model for Fault-Tolerance in Distributed Systems

Brahim Hamid and Mohamed Mosbah

LaBRI, ENSEIRB - University of Bordeaux-1,
F-33405 Talence Cedex, France
{hamid, mosbah}@labri.fr

Abstract. We present a formal method based on graph rewriting systems for the specifications and the proofs of fault-tolerant distributed algorithms. Our method deals with crash failures. In a crash failure system the process can fail by crashing, i.e. by permanently halting. The faulty processes are the processes contaminated by the crashes. The methodology is formalized in two phases. In the first phase, we build the set of illegitimate configurations to specify the faults and the faulty processes. The second phase is devoted to the addition of correction rules in the initial graph rewriting system used to encode the distributed algorithm. These rules are able to detect and eliminate the faults locally during the computation. This method can be implemented under an asynchronous message passing system which notifies the faults. To illustrate this approach, we present examples of fault-tolerant distributed spanning tree algorithms.

Keywords: Distributed systems, Fault-tolerance, Graph rewriting systems, Local computations.

1 Introduction

Distributed computing systems are becoming larger and larger, heterogeneous and complex. Since the applications running on these systems require the cooperation of many components, they are prone to faults and errors of many different types, leading to inconsistent executions. Most research works refer to two paradigms that are *self-stabilization* and *fault-tolerance*. In the first one, failures are transient [5,9] and can affect all the processes of the system. The second paradigm deals with the permanent failures. In [9] we already presented a method to design self-stabilizing distributed algorithms. In this study we focus on the permanent failures, called *crash failures*. The process which crashes is assumed to stop its execution. There are two principal approaches to improve the reliability of a system. The first is called *fault prevention* [14] and the second approach is *fault-tolerance* [2,18,3]. The aim of this approach is to provide a service in spite of the presence of faults in the system.

In a distributed system modeled by a graph, where nodes represent processes and edges communication links, a configuration is a pair (S, M), where S is a set of states of all processes and M is a set of messages that are not delivered to their receivers. We say that a configuration is correct if it is reachable from the initial states of all processes and all links are free. Informally, fault-tolerant algorithms ensure that after any failure, the system will automatically recover to reach a correct configuration in a finite time. An algorithm is called fault-tolerant if it eventually starts to behave correctly regardless of the configuration with fault components. Because the paradigm of designing fault-tolerant distributed algorithms is challenging and exciting, we are interested to study and design fault-tolerant algorithms in our framework: local computations [15]. This a powerful model to encode distributed algorithms.

The motivations of this work are on the one hand to formulate the properties of fault-tolerant algorithms by using those of rewriting systems yielding simple and formal proofs. On the other hand, as locality is an important feature of distributed computing, it is important to understand how to carry on local computations in the presence of faults.

Many fault-tolerant algorithms have been already designed [1,8,17,11,12]. However, most of these works propose global solutions which require to involve the entire system. As networks grow fast, detecting and correcting errors globally is no longer feasible. The solutions that deal with local detection and correction are rather essential because they are scalable and can be deployed even for large and evolving networks. Moreover, it is useful to have the correct (non faulty) parts of the network operating normally while recovering locally the faulty components. An important result states that consensus is impossible in asynchronous message passing system with one crashed process [7]. The basic approaches to solve this problem are to introduce some weak forms of synchrony [6] or to limit the number of crashes [13]. Therefore, such a system is improved by detection service [4,10] and consensus problem can be solved in a fault-tolerant manner.

In this work, we deal with the problem of designing algorithms encoded by local computations on a distributed computing with crash faults. We consider an asynchronous system whose processes communicate by message passing. In our approach, the properties of the program in the absence of faults are encoded by a rewriting system, and the fault-tolerance properties of the program are described with the behavior of the program when some faults occur. The faults are specified as a set of illegitimate configurations that disturb the state of the program after crashes of some component. We propose an operational and practical methodology to construct fault-tolerant protocols. A fault-tolerant distributed system is thus the system which encodes the distributed computation in a reliable system (without faults) improved by some correction rules. Those rules consist to detect and eliminate all the illegitimate configurations. This methodology is illustrated by an example of a distributed spanning tree construction.

The paper is organized as follows. The model of distributed system and the model to encode distributed algorithms are explained in Section 2. In Section 3 we present the local computations with illegitimate configurations, an extending model to represent the faulty processes. Then, we describe our method to design fault-tolerant systems. Section 4 presents example of fault-tolerant spanning tree algorithm, an application of our approach. Finally Section 5 concludes the paper.

2 Preliminaries

2.1 The Model of Distributed System

A distributed computing is modeled by a graph $G = (V, E)$, where V is a set of nodes and E is the set of edges. Nodes represent processes and edges represent bidirectional communication links. The network is anonymous, processes communicate and synchronize by sending and receiving messages through the links. The graph is unspecified and each node communicates only with its neighbors. A process can fail by crashing, i.e. by permanently halting. Communication links are assumed to be reliable. After a node fails, an underlying protocol notifies all neighbors of this node about the failure. We assume the existence of a distinguished node which is usually not crashed. A graph is $k - connected$ if the graph remains connected after the deletion of any set of $(k-1)$ nodes. The graph required is assumed to remain connected during the whole execution, we allow at most $(k-1)$ failing processes at the same time in the $k - connected$ graph. The connection of the graph guarantees that each no-crashed process can send a message to all other no-crashed process. We are interested to study the fault-tolerant distributed algorithms where the connection of the graph guarantees the existence of solution. The parameter $(k-1)$ is the degree of fault-tolerance [1] of these algorithms. We encode the fault-tolerant algorithms by graph relabeling systems [15]. As we shall see, this will simplify the proofs of the algorithms.

2.2 Graph Rewriting Systems (GRS) to Encode Distributed Algorithms

Local computations, and particularly graph relabeling systems [15] are a powerful model which provides general tools to encode distributed algorithms, to prove their correctness and to understand their power. In such a model we consider a network of processes with arbitrary topology represented as a connected, undirected graph where nodes denote processes, and edges denote communication links. Every time, each node and each edge is in some particular state and this state will be encoded by a node label or an edge label. According to its own state and to the states of its neighbors, each node may decide to realize an elementary *computation step.* After this step, the states of this node, of its neighbors and of the corresponding edges may have changed according to some specific *computation rules.* Let us recall that graph relabeling systems satisfy the following requirements:

(C1) they do not change the underlying graph but only the labeling of its components (edges and/or nodes), the final labeling being the result,

(C2) they are local, that is, each relabeling changes only a connected subgraph of a fixed size in the underlying graph,

(C3) they are locally generated, that is, the applicability condition of the relabeling only depends on the local context of the relabeled subgraph.

Let L be an alphabet and let G be a graph. We denote by (G, λ) a graph G with a relabeling function $\lambda : V(G) \cup E(G) \to L$. A graph relabeling system is a triple $\Re = (L, I, P)$ where L is a set of labels, I is a subset of L called the set of initial labels and P a finite set of relabeling rules. Consider an arbitrary system $\Re = (L, I, P)$ and a labeling function λ. A relabeling step will be denoted by $(G, \lambda) \xrightarrow[\mathcal{R}]{} (G, \lambda')$. The notion of computation then corresponds to the notion of relabeling sequence. A relabeling sequence will be denoted by $(G, \lambda) \xrightarrow[\mathcal{R}]{*} (G, \lambda')$. A graph relabeling system $\mathcal{R}$ is *noetherian* if there is no infinite $\mathcal{R}$-relabeling sequence starting from a graph with initial labels in I. We use the following notations:

(*i*) $\lambda(u)$: the labels of node u
(*ii*) $\lambda(u, v)$: the labels of the edge connecting the node u and the node v
(*iii*) $\mathcal{B}(u)$: the set of the neighbors of node u.

The program is encoded with graph relabeling system $\Re = (L, I, P)$. The labels of each process represent the value of its variables. Each rule in the set P is an action which has the following form:

$$R1 : \mathbf{RuleN}\{Precondition\}\{Relabeling\}$$

The label **R1** is the number of the rule and the label *RuleN* is the name of the action. The component *Precondition* of a rule in the program of v_0 is a boolean expression involving the labels of v_0 and the labels of its neighbors. The *Relabeling* component of a rule of v_0 updates one or more labels of v_0 and its neighbors. A rule can be executed only if its precondition evaluates *true*. The rules are atomically executed, meaning that the evaluation of a precondition and the execution of a corresponding relabeling, if the precondition is *true*, are done in one atomic step.

3 Fault-Tolerant Graph Relabeling Systems

In our model, processes can fail by crashing. The crash failures are permanent. After the crashes of some components in the distributed system, some other components become transiently faulty. We use the following definitions:

(a) *Crashed process*: a process permanently stops after a crash. It does not follow its algorithm.
(b) *Faulty process*: a process which is contaminated by a crashed process. It follows its algorithm but may deviate from that prescribed by its algorithm.
(c) *Correct process*: a process which does not belong to the set of crashed processes nor to set of faulty processes. It follows its algorithm.

From previous definitions, *fault-tolerance* is the mechanism to recover the faults (errors) introduced after the crash of some components during the computation in the distributed systems. The contamination processes are the processes which do not respect the specification of the system after the crashes occurred. These processes are the neighbors of crashed processes and we are interested to eliminate locally these bad (illegitimate) configurations.

A configuration is a pair (S, M) where S is the set of states of all processes and M is a set of messages that are not delivered to their receivers. A *local configuration* of a process is composed by its state, the states of its neighbors and the states of its communication links. In this work, we will be interested in local illegitimate configurations. To this end, we introduce a particular type of graph relabeling systems.

3.1 Graph Relabeling Systems with Illegitimate Configurations(GRSIC)

Local configurations will be defined on balls of radius 1. A *star-graph* is a rooted tree where all nodes except perhaps the root have degree 1. The root will be called the center of the star-graph. Since any ball of radius 1 is isomorphic to a star-graph, illegitimate configurations will be described through their supports (the labeled star-graphs). More precisely, an illegitimate configuration f is a labeled star-graph, say (B_f, λ_f), where B_f is a star-graph and λ_f a labeling function defined on it. Sometimes, it is useful to express such a configuration by a predicate on the edges, nodes and labels of the corresponding star-graph. For instance, a graph consisting of two nodes, u labeled A and v labeled B which are connected by an edge labeled C will be written:

$$\lambda(v) = A \ and \ \exists \ u \in \mathcal{B}(v) : \lambda(u, v) = C \ and \ \lambda(u) = B$$

For a labeled graph (G, λ), we say that a local configuration $f = (B_f, \lambda_f)$ is illegitimate for (G, λ), if there is no subgraph in (G, λ) which is isomorphic to f. In other words, there is no ball (neither sub-ball) of radius 1 in G which has the same labeling as f. This will be denoted by $(G, \lambda) \neg \vdash f$. Moreover, if $\mathcal{F}$ is a set of illegitimate configurations, we extend the last notations to $(G, \lambda) \neg \vdash \mathcal{F}$ meaning that each element of $\mathcal{F}$ is an illegitimate configuration. It means that a labeled graph (G, λ') contains an illegitimate configuration if it does not exist a labeled graph (G, λ) where:$(G, \lambda) \xrightarrow[\Re]{k} (G, \lambda')$. The parameter k is a finite number of relabeling rules' application.

A graph relabeling system with illegitimate configuration is a quadruple $\Re = (L, I, P, \mathcal{F})$ where L is a set of labels, I is a subset of L called the set of initial labels, P is a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate configurations. Let us give two examples of illegitimate configurations. Consider the following graph relabeling system given to encode a distributed spanning tree.

Assume that a unique given process is in an "active" state (encoded by label A), all other processes being in some "neutral" state (label N) . The tree initially contains the unique active node. At any step of the computation, an active node may activate one of its neutral neighbors. This computation stops as soon as all the processes have been activated. The spanning tree is then obtained by considering all the activated nodes and the edge between each node and its activated node. Every process v_i maintains two variables:

• *span*(v_i): is a variable which can have two values:
 A: v_i is in the tree
 N: v_i is not yet in the tree
• *par*(v_i): is the port number of the parent of v_i in the spanning tree, i.e the node which activated v_i.

An elementary step in this computation may be depicted as a *relabeling step* by means of the relabeling rule $R1$, given in the following, which describes the corresponding label modifications (remember that labels describe process status):

R1 : **Spanning rule**
 Precondition :
 • $\lambda(v_0) = (span(v_0), par(v_0))$
 • $span(v_0) = N$
 • $\exists\ v_i \in \mathcal{B}(v_0)$, $span(v_i) = A$
 Relabeling :
 • $span(v_0) := A$
 • $par(v_0) := v_i$

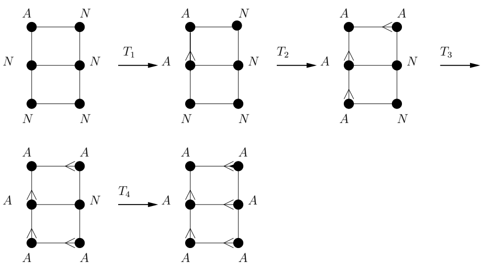


Fig. 1. Example of a distributed spanning tree's computation

Whenever an N-labeled node finds one of its neighbors labeled A, then the corresponding subgraph may rewrite itself according to the rule. After the application of the relabeling rule, node v_0 labeled $(N, 0)$ changes its label to (A, v_i) where v_i is its neighbor labeled A. A sample computation using this rule is given in Fig 1. In this figure, the value of the variable *span*(u) is the label associated to the node. The value of the variable *par*(u) is shown by $\uparrow$. Relabeling steps *may* occur concurrently on disjoint parts on the graph. The set E_m is the set of edges $(v_i, par(v_i))\ \forall\ v_i \in V$. When the graph is irreducible, i.e no rule can be applied, a spanning tree of a graph $G = (V, E)$ is computed. This tree is the graph $G_t = (V, E_m)$ consisting of the nodes of G and the set of the marked edges.

The previous algorithm can be encoded by the relabeling system $\Re_1 = (L_1, I_1, P_1)$ defined by $L_1 = \{\{N, A\} \times \{\mathbb{N}\}\}$, $I_1 = \{\{N\} \times \{0\}\}$ and $P_1 = \{R_1\}$.

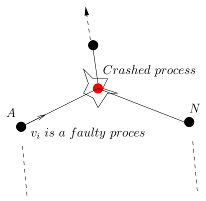


Fig. 2. The faulty process with the crashed process

Clearly, a node labeled A must have a parent, if $span(v_i) = A$ and v_i is not root, then there exists at least one neighbor of v_i labeled A, or a parent of v_i is crashed and v_i is a faulty process as shown in Fig 2. Formally, we deal with the following predicate f_1 : $span(v) = A$, $v \neq$ root *and* $\neg\exists\ u \in \mathcal{B}(v) : span(u) = A$.

3.2 Local Fault-Tolerant Graph Relabeling Systems (LFTGRS)

A local fault-tolerant graph relabeling system is a triple $\Re = (L, \mathcal{P}, \mathcal{F})$ where L is a set of labels, $\mathcal{P}$ a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate local configurations. A local fault-tolerant graph relabeling system must satisfy the two following properties:

- *Closure* : $\forall (G, \lambda) \in \mathcal{G}_{\mathcal{L}}$, if $(G, \lambda) \neg \vdash \mathcal{F}$ then $\forall (G, \lambda')$ $/(G, \lambda) \xrightarrow[\Re]{*} (G, \lambda') : (G, \lambda') \neg \vdash \mathcal{F}$
- *Convergence* : $\forall (G, \lambda) \in \mathcal{G}_{\mathcal{L}}, \exists$ an integer l : $(G, \lambda) \xrightarrow[\Re]{l} (G, \lambda') : (G, \lambda') \neg \vdash \mathcal{F}$

As for fault-tolerant algorithms, the closure property stipulates the correctness of the relabeling system. A computation beginning in a correct state remains correct until the terminal state. The convergence however provides the ability of the relabeling system to recover automatically within a finite time (finite sequence of relabeling steps). The graph required is assumed to remain connected during the whole execution, we allow at most $(k-1)$ failing processes at the same time in the $k-$*connected* graph. The connection of the graph guarantees the existence of a solution after the crashes in the graph. Consider a problem of distributed spanning tree. In our example, the existence of a spanning tree of a graph G is assured by the connection of the graph G.

As we shall see, the set of relabeling rules $\mathcal{P}$ is composed by the set of relabeling rules P used for the computation and some correction rules P_c that are introduced in order to eliminate the illegitimate configurations. The latter rules have higher priority than the former in order to eliminate faults before continuing computation.

Theorem 1. *If $\Re = (L, I, P, \mathcal{F})$ is a graph relabeling system with illegitimate configurations (GRSIC) then it can be transformed into an equivalent local fault-tolerant graph relabeling system (LFTGRS) $\Re_s = (L, P_s, \mathcal{F})$.*

Proof. We will show how to construct $\Re_s = (L, P_s, \mathcal{F})$. It is a relabeling system with priorities. To each illegitimate local configuration $(B_f, \lambda_f) \in \mathcal{F}$, we add to the set of relabeling rules the rule $R_c = (B_f, \lambda_f, \lambda_i)$ where λ_i is a relabeling function associating an initial label to each node and edge of B_f. The last relabeling function depends on the application; for example, the initial value of a node label is N in general, and the label of an edge is 0. The rule R_c is, in fact, a correction rule. Thus the set of P_s consists of the set P to which is added the set of all correction rules (one rule for each illegitimate configuration). Finally, we give a higher priority to the correction rules than those of P, in order to correct the configurations before applying the rules of the main algorithm. It remains to prove that it is a fault-tolerant system.

- *Closure*: Let $(G, \lambda) \neg \vdash \mathcal{F}$. If (G, λ') is an irreducible graph obtained from (G, λ) by applying only the rule of P, then (G, λ') does not contain an illegitimate configuration. This can be shown by induction on the sequences of relabeling steps [16,15].
- *Convergence*: Let $\mathcal{G}_{\mathcal{L}}$ be the set of graphs G and $h : \mathcal{G}_{\mathcal{L}} \longrightarrow \mathbb{N}$ be an application associating to each graph G, the number of its illegitimate configurations, then for a graph (G, λ), we have the following properties:
  . The application of a correction rule decreases $h(G)$.
  . The application of a rule in P does not increase $h(G)$.

Since, the correction rules have higher priority than the rules in P, and since the function h is decreasing, then it will reach 0 after a finite number of relabeling steps. □

Note that the last property of convergence can also be proved by using the fact that the relabeling system induced by the correction rules is noetherian. Let us note that the correction rules depend on the application. While the proofs above are based on the local reset (to the initial state) which can be heavy because it may induce a global reset by erasing all the computations, it is more efficient for particular applications to choose suitable corrections as we shall see in the following examples.

We present in the sequel a spanning tree computed by a local fault-tolerant graph relabeling system. We start by defining some illegitimate configurations to construct a set $\mathcal{F}_1$, then we improve the system by adding the correction rules to detect and eliminate these configurations. For the present system, we deal with the set $\mathcal{F}_1$ defined bellow.

Definition 1 (correct node (faulty)). *A node v is correct (resp. faulty) if it satisfies one (resp. it satisfies none) of the following properties:*

1. *if v is labeled $(A, 0)$ then $v = root$,*
2. *if v is labeled (A, u) then there exists one node u labeled (A, w),*
3. *if v is labeled $(N, 0)$ then there does not exist node u labeled (A, v).*

From Definition 1, $\mathcal{F}_1 = \{f_1, f_2\}$, where f_1 and f_2 are defined as :
f_1 : $\exists\ v_0 \neq$ root, $span(v_0) = A$ and $\neg\exists\ v_i \in \mathcal{B}(v_0) : par(v_0) = v_i$ *and* $span(v_i) = A$.
f_2 : $\exists\ v_0$, $span(v_0) = A$, $par(v_0) = v_i$ and $span(v_i) = N$.

The correction rules are deduced from the previous configurations:

Rc1 : **Crash of a parent rule**

Precondition :

- $v_0 \neq$ root
- $\lambda(v_0) = (span(v_0), par(v_0))$
- $span(v_0) = A$
- $\neg\exists\ v_i \in \mathcal{B}(v_0) : par(v_0) = v_i$ *and* $span(v_i) = A$

Relabeling :

- $span(v_0) := N$
- $par(v_0) := 0$

Rc2 : **Cleaning rule**

Precondition :

- $\lambda(v_0) = (span(v_0), par(v_0))$
- $span(v_0) = A$
- $par(v_0) = v_i$
- $span(v_i) = N$

Relabeling :

- $span(v_0) := N$
- $par(v_0) := 0$

We assume in this system the existence of a distinguished node called the root which is initially labeled A and which is usually correct.

We define the relabeling system $\Re_{s_1} = (L_1, P_{s_1}, \mathcal{F}_1)$, where $P_{s_1} = \{R1, Rc1, Rc2\}$ such that $Rc1, Rc2 \succ R1$. We now state the main result.

Theorem 2. *The relabeling system $\Re_{s_1}$ is locally fault-tolerant. It encodes a fault-tolerant distributed algorithm to compute a spanning tree.*

Proof. The proof of fault-tolerance results from Theorem 1. To show that the result is a spanning tree, we use the following invariants which can be proved by induction on the size of the relabeling sequences:

(I1) All N-labeled nodes are 0-parent.
(I2) Each parent is an A-labeled node.
(I3) The subgraph induced by the *node-parent* edges is a tree.
(I4) The obtained tree of the irreducible graph is a spanning tree. □

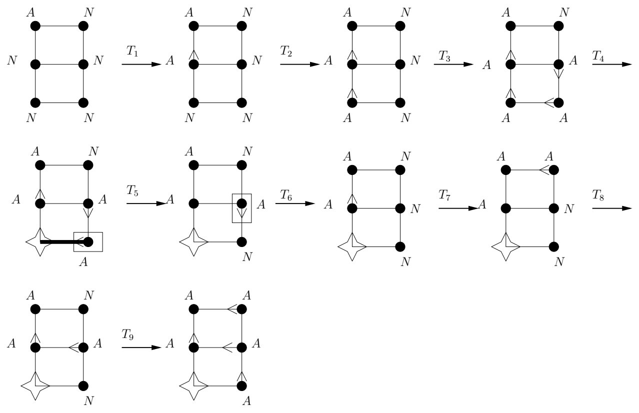


Fig. 3. Example of fault-tolerant spanning tree algorithm execution

Fig. 3 gives a sample computation of a spanning tree with a crash of a process after the step T_4. The steps T_1, T_2 and T_3 represent the application of the main rule $R1$. The process corresponding to the node shown by a star crashes after the step T_4, and remains in a faulty state until the end of the execution. Since the edge incident to this node belongs to the spanning tree (bold edge), it must be deleted from the tree and the adjacent node will be labeled N. That is done in step T_5 which is an application of $Rc1$ by the node in the square. Now, the latter node labeled N is a parent of a node labeled A. In step T_6, the node in the square applies the rule $Rc2$ by relabeling itself to N. Note that since $Rc1$ and $Rc2$ have highest priority, it will be applied on the context of the faulty node before $R1$. Then, in step T_7, T_8, T_9, the rule $R1$ is applied allowing to continue the computation of the spanning tree by avoiding the faulty node.

4 Example: Spanning Tree with Termination Detection

Let us illustrate fault-tolerant distributed algorithm which computes a spanning tree of a network with termination detection. We start with an algorithm in a network without crashes.

Assume that a unique given process called the "root" is in an "active" state (encoded by label (A,0)), all other processes being in some "neutral" state (label (N,0)). The tree initially contains the unique active node. At any step of the computation, an active node may activate one of its neutral neighbors. Then the neutral neighbor becomes active and marks with the variable *par* its activated neighbor. When node v_0 cannot activate any neighbor because all of these have already been activated by some other nodes, v_0 transforms its state into a "feedback" state. When all the activated nodes("sons") of v_0 are in the "feedback" state, it transforms its state into a "feedback" state. The root detects the termination of the algorithm when it is in the "feedback" state. Every process v_i maintains two variables:

- *span*(v_i): is a variable which takes three values:
  N: v_i is not yet in the tree
  A: v_i is in the tree
  F: v_i is in the feedback state, it finds all its neighbors in the tree and all its sons in the feedback state
  T: the termination detection at the root

- par_i: is the number of the port connected v_i to its activated neighbors

We consider the following relabeling system which encodes a distributed algorithm computing a spanning tree with termination detection, $\Re_2 = (L_2, I_2, P_2)$ defined as $L_2 = \{\{N, A, F, T\} \times \{\mathbb{N}\}\}$, $I_2 = \{\{N\} \times \{0\}\}$, $P_2 = \{R1, R2, R3, R4\}$. The label of each node v_0 is $(span(v_0), par(v_0))$. Now we present the set of rules:

R1 : **Root diffusion rule**

Precondition :
- $\lambda(v_0) = (span(v_0), par(v_0))$
- v_0 =root
- $span(v_i) = N$

Relabeling :
- $span(v_0) := A$

R2 : **Node diffusion rule**

Precondition :
- $\lambda(v_0) = (span(v_0), par(v_0))$
- $span(v_0) = N$
- $\exists\ v_i \in \mathcal{B}(v_0)$, $span(v_i) = A$

Relabeling :
- $span(v_0) := A$
- $par(v_0) := v_i$

R3 : **Node feedback rule**

Precondition :
- $\lambda(v_0) = (span(v_0), par(v_0))$
- $v_0 \neq$ root
- $span(v_0) = A$
- $\forall\ v_i \in \mathcal{B}(v_0) : (span(v_i) \neq N)$ *and* $(par(v_i) \neq v_0$ *or* $span(v_i) = F)$

Relabeling :
- $span(v_0) := F$

R4 : **Root detection of termination rule**

Precondition :
- $\lambda(v_0) = (span(v_0), par(v_0))$
- $v_0 =$ root
- $span(v_0) = A$
- $\forall\ v_i \in \mathcal{B}(v_0) : span(v_i) = F$

Relabeling :
- $span(v_0) := T$

We present in the sequel a spanning tree with termination detection computed by a local fault-tolerant relabeling system. We start by defining some illegitimate configurations to construct a set $\mathcal{F}_2$, then we improve the system by adding the correction rules to detect and eliminate these configurations.

Note that we distinguish between crashed node and faulty node as explained in the previous section. A faulty node should be viewed as one which has to reconstruct the computation because of the crash of some other nodes.

Definition 2 (correct node (faulty)). *A node v is correct (resp. faulty) if it satisfies one (resp. it satisfies none) of the following properties:*

1. *if $span(v) \in \{A, F, T\}$ and $par(v) = 0$ then $v = root$,*
2. *if v is labeled (A, u) then there exists one node u labeled (A, w),*
3. *if v is labeled (F, u) then there exists one node u labeled (l, w), where $l \in \{A, F\}$,*
4. *if v is labeled $(N, 0)$ then there does not exist node u labeled (l, v), where $l \in \{A, F\}$.*

For the present system, we deal with the following set $\mathcal{F}_2 = \{f_1, f_2, f_3\}$ where f_1, f_2 and f_3 are:
$f_1 : \exists\ v_0 \neq \text{root},\ span(v_0) = A$ and $\neg\exists\ v_i \in \mathcal{B}(v_0) : par(v_0) = v_i$ *and* $span(v_i) = A$.
$f_2 : \exists\ v_0 \neq \text{root},\ span(v_0) = F$ and $\neg\exists\ v_i \in \mathcal{B}(v_0) : par(v_0) = v_i$ *and* $span(v_i) \in \{A, F\}$.
$f_3 : \exists\ v_0,\ span(v_0) \in \{A, F\},\ par(v_0) = v_i$ and $span(v_i) = N$.

The correction rules are deduced from the previous configurations:

Rc1 : **Crash of a parent rule 1**

Precondition :

- $v_0 \neq$ root
- $\lambda(v_0) = (span(v_0), par(v_0))$
- $span(v_0) = A$
- $\neg\exists\ v_i \in \mathcal{B}(v_0) : par(v_0) = v_i$ *and* $span(v_i) = A$

Relabeling :

- $span(v_0) := N$
- $par(v_0) := 0$

Rc2 : **Crash of a parent rule 2**

Precondition :

- $v_0 \neq$ root
- $\lambda(v_0) = (span(v_0), par(v_0))$
- $span(v_0) = F$
- $\neg\exists\ v_i \in \mathcal{B}(v_0) : par(v_0) = v_i$ *and* $span(v_i) \in \{A, F, T\}$

Relabeling :

- $span(v_0) := N$
- $par(v_0) := 0$

Rc3 : **Cleaning rule**

Precondition :

- $\lambda(v_0) = (span(v_0), par(v_0))$
- $span(v_0) \in \{A, F\}$
- $par(v_0) = v_i$
- $span(v_i) = N$

Relabeling :

- $span(v_0) := N$
- $par(v_0) := 0$

We assume in this system that the "root" is usually correct. We define the relabeling system $\Re_{s_2} = (L_2, P_{s_2}, \mathcal{F}_2)$, where $L_2 = \{\{N, A, F, T\} \times \{\mathbb{N}\}\}$ and $P_{s_2} = \{R1, R2, R3, Rc1, Rc2, Rc3\}$ such that $Rcj \succ Ri$. We now state the main result:

Theorem 3. *The relabeling system $\Re_{s_2}$ is locally fault-tolerant. It encodes a fault-tolerant distributed algorithm to compute a spanning tree with termination detection.*

Proof. The proof of local fault-tolerant results from Theorem 1. To show that the result is a spanning tree, it suffices to use invariants like those of the preceding example. □

5 Conclusion

We have presented a method to design fault-tolerant algorithms encoded by local computations. The method consists of specifying a set of illegitimate configurations to describe the faults that can occur during the computation, then adding local correction rules to the corresponding algorithm which is designed in a safe mode. These specific rules are of high priority and are performed in order to eliminate the faults that are detected locally. We introduce and illustrate this approach with a distributed spanning tree algorithms in k-connected graph which allows to tolerate until $(k-1)$ crashed faulty processes.

Our approach can be applied in practical applications as a generic and automatic method to deal with faults in distributed systems. For instance, in a very large network, assume that the diffusion of messages between sites is performed using a spanning tree of the network. Now, if some central node crashes, then our method allows to find a solution to continue the diffusion service. We are currently working on applying our solution to particular architectures and mainly web services.

References

1. E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. In *WDAG93: Distributed Algorithms 7th International Workshop Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, 1993.
2. A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Softw. Eng.*, 19(11):1015–1027, 1993.

3. P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst.*, 26(1):125–185, 2004.
4. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed system. *Journal of the ACM*, 43(2):225–267, July 1996.
5. E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
6. M.J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. In *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 59–70. ACM Press, 1985.
7. M.J. Fisher, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
8. F. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.
9. B. Hamid and M. Mosbah. An automatic approach to self-stabilization. In *6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD2005), Baltimore, USA (to appear)*, pages 129–132, May 2005.
10. B. Hamid and M. Mosbah. An implementation of a failure detector for local computations in graphs. In *Proccedings of the 23rd IASTED International multi-conference on parallel and distributed computing and networks*, February 2005.
11. S.S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93. Springer-Verlag, 2000.
12. S. Kutten and D. Peleg. Tight fault locality. *SIAM J. Comput.*, 30(1):247–268, 2000.
13. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
14. J. C. Laprie. *Dependability—Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-tolerant Systems*. Springer-Verlag, 1992. IFIP WG 10.4.
15. I. Litovsky, Y. Mtivier, and E. Sopena. Graph relabeling systems and distributed algorithms. In World Scientific Publishing, editor, *Handbook of graph grammars and computing by graph transformation*, volume Vol. III, Eds. H. Ehrig, H.J. Kreowski, U. Montanari and G. Rozenberg, pages 1–56, 1999.
16. Y. Mtivier, M. Mosbah, and A. Sellami. Proving distributed algorithmes by graph relabeling systems: Example of tree in networks with processor identities. *In applied Graph Transformations (AGT2002), Grenoble*, April 2002.
17. A. Porat. Maintenance of a spanning tree in dynamic networks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, page 282. ACM Press, 1999.
18. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier*

Anjali Joshi and Mats P.E. Heimdahl

Department of Computer Science and Engineering,
University of Minnesota, 200 Union St SE,
Minneapolis, MN 55455, USA
Phone: 1 612 624 7590, Fax: 1 612 625 0572
{ajoshi, heimdahl}@cs.umn.edu

Abstract. Safety analysis techniques have traditionally been performed manually by the safety engineers. Since these analyses are based on an informal model of the system, it is unlikely that these analyses will be complete, consistent, and error-free. Using precise formal models of the system as the basis of the analysis may help reduce errors and provide a more thorough analysis. Further, these models allow automated analysis, which may reduce the manual effort required.

The process of creating system models suitable for safety analysis closely parallels the model-based development process that is increasingly used for critical system and software development. By leveraging the existing tools and techniques, we can create formal safety models using tools that are familiar to engineers and we can use the static analysis infrastructure available for these tools. This paper reports our initial experience in using *model-based safety analysis* on an example system taken from the ARP Safety Assessment guidelines document.

1 Introduction

Traditionally, safety engineers manually perform analyses, such as fault tree analysis, based on informal design models and requirements documentation. Unfortunately, these analyses are highly subjective and dependent on the skill of the practitioner. We hypothesize that by redirecting the effort to build models of the system under study and its fault model we can both reduce the effort involved and increase the quality of the analysis. To this end, we propose a *model-based safety analysis* process in which engineers create formal models for both the system design and safety analysis, and use automated analysis tools to analyze their behavior. We describe our early experience towards this goal in this paper.

Our approach is to adapt model-based development techniques using formal modeling languages and tools such as SCADE [9] and Simulink [5] for safety analysis. By integrating these tools into safety analysis, it is possible to create system models that can be simulated and analyzed using a variety of static

* This work has been partially supported by NASA contract NCC-01-001.

analysis techniques. This combination allows an analyst to quickly explore different "what-if" scenarios on combinations of faults using simulation, and also allows formal verification of different aspects of fault tolerance and, potentially, autogeneration of safety analysis artifacts such as fault trees.

We describe our preliminary experiences using model-based safety analysis with a wheel brake system example adopted from ARP 4761 [1], a standards document for safety analysis in the avionics industry. With the help of this example, we illustrate how we can derive benefits from a model-based safety analysis in a practical setting using existing tools. At the same time, this exercise exposes several issues and shortcomings that need to be addressed to make formal safety analysis acceptable in practice.

2 Safety Assessment Process

The overall safety assessment process that is followed in practice in the avionics industry is described in the SAE standard ARP 4761 [1]. Our summary in this section is largely adopted from ARP 4761.

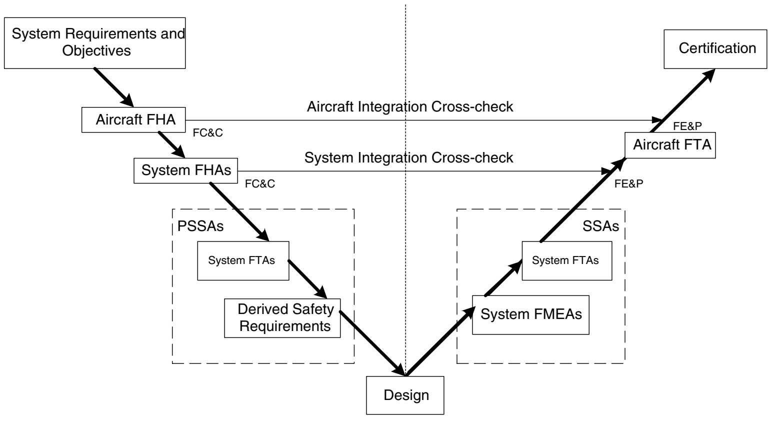

System Requirements and Objectives
Aircraft FHA
FC&C
System FHAs
FC&C
Aircraft Integration Cross-check
System Integration Cross-check
PSSAs
System FTAs
Derived Safety Requirements
Design
SSAs
System FMEAs
System FTAs
FE&P
Aircraft FTA
FE&P
Certification

Fig. 1. Traditional "V" Safety Assessment Process

Figure 1 shows an overview of the safety assessment process as recommended in ARP 4761. The process includes safety requirements identification (the left side of the "V" diagram) and verification (the right side of the "V" diagram), that support the aircraft development activities. An aircraft level Functional Hazard Analysis (FHA) is conducted at the beginning of the aircraft development cycle, which is then followed by system level FHA for individual sub-systems. The FHA is followed by Preliminary System Safety Assessment (PSSA), which derives safety requirements for the subsystems, primarily using Fault Tree Analysis (FTA). The PSSA process iterates with the design evolution, with design changes necessitating changes to the derived system requirements (and also to the fault trees) and potential safety problems identified through the PSSA leading to design changes. Once design and implementation are completed, the System Safety Assessment (SSA) process verifies whether the safety requirements are met in the implemented design. The system Failure Modes and Effects Analysis (FMEA) is performed to compute the actual failure probabilities on the items. The verification is then completed through quantitative and qualitative analysis of the fault trees created for the implemented design, first for the subsystems and then for the integrated aircraft.

We propose to modify this traditional "V" process so that the lower level PSSA and SSA activities are performed based on a formal model of the system under consideration. Figure 2 shows the modified "V" diagram for model-based safety analysis. The shaded blocks are those activities that will be modified or added.

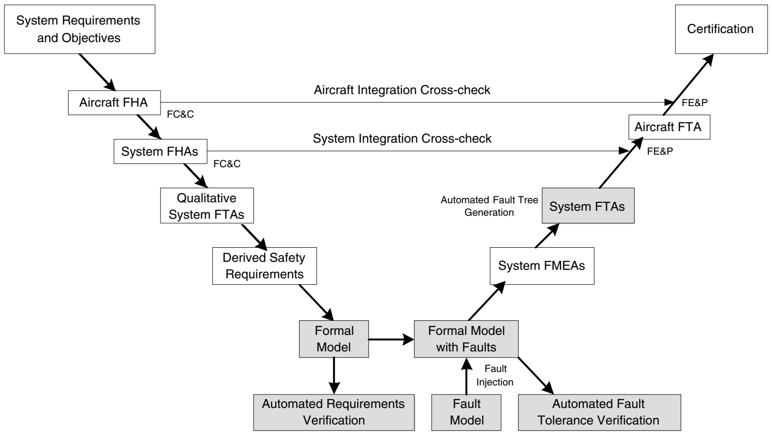

System Requirements and Objectives
Certification
Aircraft Integration Cross-check
Aircraft FHA
FC&C
FE&P
Aircraft FTA
System Integration Cross-check
System FHAs
FC&C
FE&P
Qualitative System FTAs
Automated Fault Tree Generation
System FTAs
Derived Safety Requirements
System FMEAs
Formal Model
Formal Model with Faults
Fault Injection
Automated Requirements Verification
Fault Model
Automated Fault Tolerance Verification

Fig. 2. Modified "V" Safety Assessment Process

As we can observe from Figure 2, the parts of the analysis that are primarily affected are at the bottom of the "V". The biggest difference is that the safety analysis activities at this level are now focused around a formal model of the system behavior, and that many of the artifacts of the safety analysis can be derived from this model. The idea is to try to pose the right verification questions to formal tools (such as model checkers and theorem provers) so that it is possible to derive the necessary safety analysis information. We then wish to turn the results of these analyses back into artifacts that can be easily understood and used by safety engineers.

3 Model-Based Safety Analysis Process

The primary step in a model-based safety analysis is creating a formal specification of the system model. The behavior of the system can be specified in formal specification languages supporting graphical and/or textual representation; e.g., synchronous (textual) languages like RSML^{-e} [10] and Lustre [6], and graphical tools like Simulink [5] and SCADE [9]. The logical and physical architecture of the system can be specified in an architecture description language.

The derived safety requirements are determined in the same way as in the traditional “V” process. To support automated analysis, the safety properties must be expressed in some formal notation. There are several candidate notations, including temporal logics like CTL/LTL or higher order predicate logics. One can also specify safety requirements as small behavioral models in some formal specification language.

To be able to apply formal verification tools to perform safety analysis, in addition to formalizing the system model, we also need to formalize the fault model. The fault model, in addition to common failure modes like *non-deterministic*, *inverted*, *stuck_at* etc, could encode information regarding fault propagation, simultaneous dependent faults and fault hierarchies, etc.

After specifying the fault model and composing it with the original system model, the safety analysis involves verifying whether the safety requirements hold in presence of the faults defined in the fault model. The safety engineer can perform exploratory analysis using formal verification tools, e.g., what is the largest n such that the particular safety requirement holds in face of n faults?. The notion could also be specialized to a specific combination of faults rather than random combinations. With adequate tool support, the formal verification results could be represented in the form of familiar safety artifacts like fault trees.

In the following sections, we illustrate some of our early results in applying the model based safety analysis process on a wheel brake system (WBS) example derived from the ARP safety analysis guidelines [1]. In section 4, we describe the informal requirements of the example. Next, in Sections 5 and 6, we describe how the system model without failures can be encoded in Simulink and how we can verify safety properties of interest on the model. In section 7, we describe a simple fault model for the WBS components and extend our system model to include component faults. Section 8 briefly describes the exploratory safety analysis performed on the extended model using the SCADE Design Verifier.

4 Wheel Brake System Example

We illustrate some of the basic activities involved in model based safety analysis with the help of an example of a Wheel Brake System (WBS), as described in ARP 4761 - Appendix L [1]. We chose this example primarily because the ARP 4761 document is used as the main reference for safety assessment by majority of the safety engineers in the avionics community.

This section consists of excerpts from the ARP 4761 document giving the informal requirements for WBS. The informal WBS diagram taken from the ARP 4761 document is shown in Figure 3. The WBS is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the aircraft during taxiing and landing phases, and in the event of a rejected take-off. Braking on the ground is either commanded manually, via brake pedals, or automatically (autobrake) without the need for pedal application. The

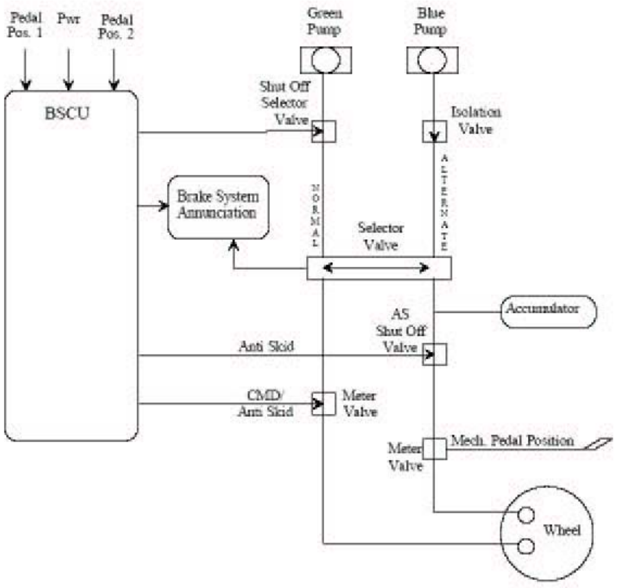


Fig. 3. Wheel Brake System as shown in ARP 47-61

Autobrake function allows the pilot to pre-arm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobreak function will control break pressure to provide a smooth and constant deceleration.

Based on the requirement that loss of all wheel braking is less probable than $5 \cdot 10^{-7}$ per flight, a design decision was made that each wheel has a brake assembly operated by two independent sets of hydraulic pistons. One set is operated from the `GREEN` pump and is used in the `NORMAL` braking mode. The `ALTERNATE` braking system is on standby and is selected automatically when the `NORMAL` system fails. The `ALTERNATE` system is supplied pressure by both the `BLUE` pump and an `ACCUMULATOR`, both of which can be used to drive the brake. The accumulator is the reserve pressure reservoir with built up pressure that can be reliably released if both of the two primary pumps (the Blue and Green pumps) fail. The accumulator drives the `ALTERNATE` system in the `EMERGENCY` braking mode.

Switch-over between the hydraulic pistons and the different pumps is automatic under various failure conditions, or can be manually selected. Reduction of `GREEN` pressure below a threshold value, either from loss of the `GREEN` pump itself or from its removal by the Break System Control Unit (BSCU) due to the presence of faults, causes an automatic selector to connect the `BLUE` supply to the `ALTERNATE` brake system. If the `BLUE` pump fails, then the `ACCUMULATOR` is used to supply hydraulic pressure.

An anti-skid facility is available in both the `NORMAL` and `ALTERNATE` system modes. The anti-skid function is similar to the anti-lock brakes common on passenger vehicles and operates largely in the same manner.

In the `NORMAL` mode, the brake pedal position is electronically provided to a braking computer. This in turn produces corresponding control signals to the brakes. In addition, the braking computer monitors various signals that denote certain critical aircraft and system states to provide correct brake functions and improve system fault tolerance, and generates warnings, indications and maintenance information to other systems.

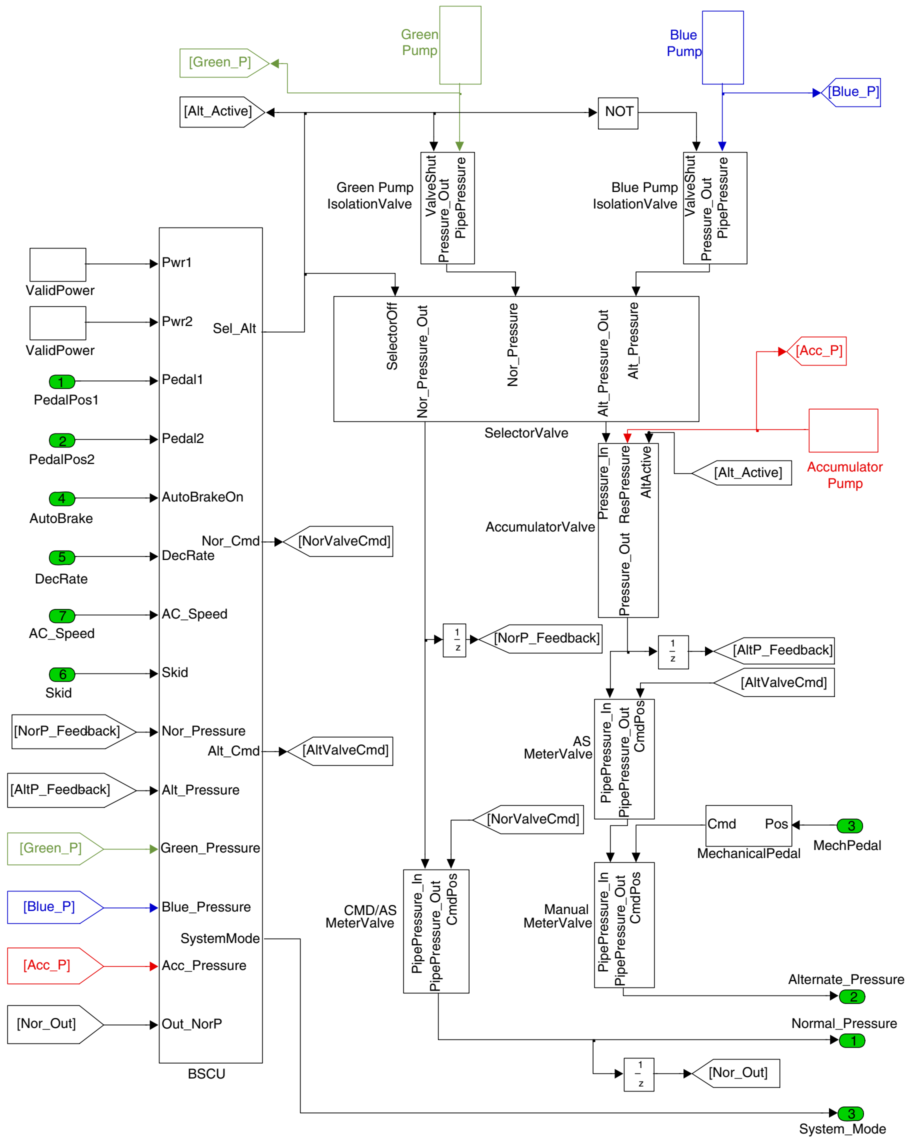


Fig. 4. Nominal Wheel Brake System in Simulink

5 Nominal Wheel Brake System in Simulink

The informal requirements of the WBS as specified in the ARP document were not found to be particularly rigorous. To implement a working model, we had to make several assumptions about the system that still need to be confirmed with the authors of ARP 4761. Figure 4 illustrates how we can model the WBS in Simulink. The model captures both digital and mechanical components of the system and reflects the informal structure of the system as given in the ARP document.

WBS (the highest level component/system) consists of a digital control unit, the BSCU, and two hydraulic pressure lines, `NORMAL` (pressured by the Green Pump) and `ALTERNATE` (pressured by the Blue Pump and the Accumulator) line. The system takes the following inputs from the environment - PedalPos1, PedalPos2, AutoBrake, DecRate, ACSpeed, Skid, and MechPedal. All of the above inputs, except MechPedal, are forwarded to the BCSU for computing the brake commands. There are also a number of mechanical components along the two hydraulic lines, for example different types of valves. We have defined a library of common components such as the MeterValve, IsolationValve, Pump, etc., which are then instantiated at various locations in the WBS. The outputs of the WBS are Normal_Pressure (hydraulic pressure at the end of the Normal line), Alternate_Pressure (hydraulic pressure at the end of the Alternate line) and System_Mode (computed by the BSCU).

Due to lack of space, we cannot describe the Simulink model in full detail[1]. To illustrate some aspects of fault modelling, we explain the implementation of the `MeterValve` component, which is used in three places in Figure 4: the `CMD/AS MeterValve` on the Normal hydraulic line and the `AS MeterValve` and `Manual MeterValve` on the Alternate hydraulic line. The meter valve implementation takes two inputs, the incoming pipe pressure and the valve position command, and generates an output pressure which depends on the valve position.

6 System Verification

After creating the system model, we would like to verify that some basic safety properties hold on the *nominal system*, an idealized system containing no faults. As a first step, we need to formalize the derived safety requirements as safety properties. Simulink does not directly support any model-checking tools, so to perform this step, we import the Simulink model into SCADE, which contains the Design Verifier model checker. The properties can be formalized in Lustre, which is the underlying textual notation for SCADE.

Throughout this paper, we use an example safety requirement that is given in the ARP 4761 document,

> *Loss of all wheel braking (unannunciated or annunciated) during landing or RTO shall be less than* $5 \cdot 10^{-7}$ *per flight.*

[1] We will publish the complete Simulink model on our web site: http://www.cs.umn.edu/crisys

Since we are not considering annunciations in this model and we are not considering any quantitative analysis at this stage, let us simplify this safety requirement and state the undesirable event we are trying to prevent as simply,

Loss of all wheel braking during landing or RTO shall not occur.

We consider that the hydraulic pressure at the output should be above some minimum constant threshold to have any effect on the braking. Recall from Section 4 that we have variables PedalPos1, PedalPos2, and MechPedal, that describe the electric and mechanical pedal positions, respectively. We can state our safety property as,

When all pedals are pressed, then either the normal pressure or the alternate pressure should be above the threshold.

We first define two intermediate variables in Lustre to represent whether all of the pedals are being pressed (AllPed) and whether any pressure is being provided to the brakes (SomePressure).

```
AllPed       = (IS_PedalPressed(PedalPos1) and IS_PedalPressed(PedalPos2)
                and IS_PedalPressed(MechPedal));
SomePressure = (Normal_Pressure > threshold) or
               (Alternate_Pressure > threshold);
```

IS_PedalPressed is a predicate that returns true when pedal is pressed. AllPed and SomePressure are then used in the property SomePressure_Property as

```
SomePressure_Property = Implies(AllPed,SomePressure);
```

We used Design Verifier in an attempt to verify this property, which was initially found to be falsifiable. If the wheels do not have traction, the anti-skid functionality will be activated and the pressure at the wheels may indeed be lowered below the threshold to allow the wheels to regain traction. Since this is expected and acceptable behavior, we modify our safety property accordingly, by extending AllPed to AllPedNoSkid, where we require that the pedals are pressed and that we are not skidding.

```
AllPedNoSkid = (IS_PedalPressed(PedalPos1) and IS_PedalPressed(PedalPos2)
                and IS_PedalPressed(MechPedal) and not (Skid));
```

Now, the SomePressure property is verified by Design Verifier: if all pedals are pressed and we are not skidding then we will have some pressure at the brakes.

7 Extension with a Fault Model

In Section 5, we created a model describing the nominal behavior of the system. To perform the safety analysis on this model, we would like to extend it to describe possible fault behavior. This section illustrates specification of the fault model and extension of the nominal model with this fault behavior in Simulink.

Failure modes are introduced in the analysis to capture the various ways in which the components of the system can malfunction. We want to be able to model both persistent and intermittent failures and also multiple simultaneous failures. Traditionally, failure modes specify predefined ways in which components can fail, e.g., the output from a digital component might be stuck at a particular value, inverted, take on a nondeterministic value (unconstrained value), etc. In the WBS example, the mechanical failures considered include different variants of stuck valves corresponding to the different kinds of valves, power failure to the BSCU, and pump failures. We also consider one digital failure mode for the BSCU component, an inverted signal for the Boolean Sel_Alt (select alternate system) output.

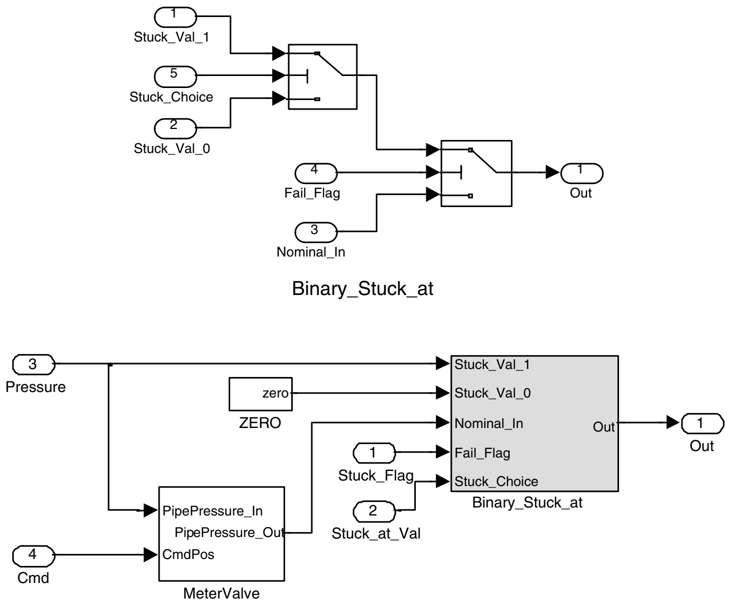


Fig. 5. Binary Stuck at failure mode and MeterValve fault extension

Let us consider the notion of a valve stuck open or closed in more detail. The manifestation of this failure must consider the original input pressure to the component (in case the valve is stuck open) and override the normal output of the valve. We create a simple fault model in which a component can either be stuck open or closed in Figure 5. Binary_Stuck_at failure mode switches between the stuck value and the nominal value depending on the boolean Fail_Flag (fault trigger). The stuck value could be either Stuck_Val_1 (open) or Stuck_Val_0 (closed) depending on the boolean Stuck_Choice. Thus, we define two 'special' outputs for the failure mode depending on whether the component is stuck open or closed; if it is not stuck, we output the nominal value of the original component. We then extend the MeterValve component to MeterValve_Stuck using this failure mode (Figure 5). When Stuck_Choice is 1 the meter valve is stuck open and the input pressure is forwarded as is to the output, ignoring the valve position command. When Stuck_Choice is 0 the valve is stuck closed and the output pressure is set to 0.

To extend the original model, the nominal mechanical components from the original model (Figure 4) are replaced by the corresponding components extended with failure modes. To control the fault behavior of the extended model, a number of fault inputs need to be added to the system. For example, all the valve components, extended by the stuck_at failure mode, have two additional inputs: Stuck_Flag and Stuck_Val. The rest of the failure modes require a single input signaling the occurrence of a fault. After extension, the model looks fairly similar to Figure 4, but adds some complexity and clutter due to the number of additional inputs necessary to describe the possible faults.

8 Exploratory Safety Analysis

After extending the model with the faults, we would like to check the fault tolerance of our system, i.e., we want to check that the system is tolerant to a certain maximum number of faults. More specifically, we would like to investigate two types of faults using this approach—*transient single step faults* and *faults lasting over an arbitrary number of steps*, which can simulate permanent faults. For this example we again formalize our safety properties in Lustre and use the SCADE Design Verifier for verification. To make it easier to specify properties, we extend our model to compute the total number of fault inputs that are true in the current step (this number given by NumFails).

First, let us verify if our safety requirement holds in the presence of one fault.

If there is one fault and all pedals are pressed in absence of skidding, then either the normal pressure or the alternate pressure should be above the threshold.

We can formalize this in Lustre as,

```
Prop_Orig = fby(Implies(((NumFails = 1) and AllPedNoSkid),
                         SomePressure), 1, true) ;
```

Lustre expressions always look at the current and past instants. Implies encodes implication and pre operator examines values of variables from previous steps. The fby operator looks at the n-th previous value of an expression (in this case, the $Implies$ expression). The second argument (1) of the fby expression is the value for n. The third argument (true) describes the value of the fby operator in the initial state.

When attempting to verify `Prop_Orig` using Design Verifier, it returns a counterexample. We realize that, due to some latency, the system cannot respond to most faults in the same step in which they occur. Unfortunately, even after extending the number of steps to respond, if our only constraint is on the number of faults, the model checker finds a counterexample. It gives a scenario in which the fault *migrates*: the system toggles between faults on the Normal line and the

Alternate line and can never recover. Since this situation is highly unlikely, we rule it out. We do so by stating that any transient fault will be followed by a few steps in which no other transient fault occurs. In other words,

> *If there is one single step fault and in the next step all pedals are pressed in absence of skidding, then in the next step either the normal pressure or the alternate pressure should be above the threshold.*

The encoding in Lustre is as follows:

```
Antecedent = pre(NumFails = 1) and AllPedNoSkid and (NumFails = 0);
Consequent = SomePressure ;
Prop_SingleStepSingleFail = fby(Implies(Antecedent, Consequent),2,true) ;
```

However, the Design Verifier still returns with a counterexample. We observe that there is an additional step delay for the system to detect failures located on the `NORMAL` system and switch to the `ALTERNATE` system. We deem this delay acceptable and modify our property again. After allowing for an additional delay in the property, the Deign Verifier verifies it. Thus, we can formally verify that our system can recover from one transient fault in at most three steps.

Now, we want to investigate how our system responds to *persistent* faults. To describe this fault scenario, we define a boolean variable, `Changed`, which takes on the value true when one of more of the fault trigger inputs change their values. Using this variable, we can describe persistent faults in which *the same* fault occurs for an arbitrary number of steps. The following property is the same as the earlier transient property, except that now we have `not(Changed)` instead of `(NumFails = 0)` to encode that the same fault persists in the following two steps.

```
Antecedent=pre(pre(NumFails = 1)) and pre(AllPedNoSkid and not(Changed))
                and AllPedNoSkid and not(Changed) ;
Consequent=pre(SomePressure) or SomePressure ;
Prop_MultiStepSingleFail = fby(Implies(Antecedent,Consequent),3,true) ;
```

Design Verifier again finds a counterexample, and from this, we observe that there is an additional delay required for the system to respond to some persistent faults, in the situation when the system is switching back to the `NORMAL` hydraulic system from the `ALTERNATE` system. In this instance, it takes an additional step to check if a persistent fault on the `NORMAL` line is still present. To handle this case, we add one additional step for the system to stabilize. Design verifier no verifies that the system will behave as expected. Thus, we verify that the system can recover from single transient or persistent faults within an acceptable time frame.

However, we can easily observe that the system is not tolerant to two (or more) simultaneous continuous failures. Design Verifier immediately comes back with a counter-example where two meter valves fail along both the normal and the alternate hydraulic lines. Note that, the safety engineer can explore different combinations of faults that the system can tolerate. There will not be even a glitch in the output pressure if all the components on the Alternate line fail when no component along the Normal line fails.

9 Related Work

Most of the work in automating safety analysis has been in automatically generating fault trees. FSAP/NuSMV-SA [4] is a tool, developed as part of the ESACS project [3], for automating the generation of fault trees. The ESACS methodology supports integrated design and safety analysis of systems. The FSAP tool requires the system model to be specified in NuSMV and has support for failure mode definition and model extension through automatic failure injection. FSAP uses the NuSMV model checker to generate a fault tree given a top level event in temporal logic. Though FSAP is a very powerful tool, it has disadvantages, which might limit its applicability to practical systems. A fault tree generated by FSAP has a flat structure; the structure of the generated fault trees is an "or-and" structure, i.e., it is a disjunction of all the minimum cut sets, with each minimum cut set being a product of basic events. A fault tree generated by a traditional manual analysis is usually more intuitive to read as the analyst creates the fault tree to correspond to the structure of the system. Also, we observed that there isn't a lot flexibility in defining the fault model - no good way of specifying fault propagation, simultaneous/dependent faults, and persistent/intermittent faults. Also, FSAP cannot describe even moderately complex faults, such as stuck_at, as it can only affect the output of a component.

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [8] [7] is a method for safety analysis that enables integrated assessment of a complex system from the functional level through to the low level of component failure modes. The failure behavior of components in the model is analyzed using a modification of classical FMEA called Interface Focused-FMEA (IF-FMEA). One of the strong points of this approach is that the fault tree synthesis algorithm neatly captures the hierarchical structure of the system in the fault tree.

The Altarica language was designed to formally specify the behavior of systems when faults occur [2]. An Altarica model can be assessed by means of complementary tools such as fault tree generator and model-checker. In terms of fault modeling, there seems to be no good support for simultaneous and dependent failures. Altarica does not differentiate between transient and permanent faults.

10 Summary and Conclusion

We describe Model-Based Safety Analysis, an approach for automating portions of the safety analysis process using executable formal models of the system. This approach is based on existing commercial tools and techniques that are increasingly used for systems and software engineering for safety-critical systems. We have modelled the Wheel Brake System example from ARP 4761 - Appendix L [2]. We illustrated how this system can be modelled and investigated for safety and fault tolerance. We believe that the model-based safety analysis approach has several benefits to offer to a next-generation safety analysis process. For instance,

– A tighter integration between systems and safety analysis based on common models of system architecture and failure modes.
– The ability to simulate the behavior of system architectures early in the development process to explore potential hazards.
– The ability to exhaustively explore all possible behaviors of a system architecture with respect to some safety property of interest using automated analysis tools.
– The ability to automatically generate many of the artifacts that are manually created during a traditional safety analysis such as fault trees and FMEA/FMECA charts.

Although we have received positive feedback from our industry partners, there are several research challenges that must be addressed before the full benefits of model-based safety analysis can be fully realized. First, there are questions as to which languages and tools are most suitable and how much modeling detail is necessary to perform useful analysis. Second, we observed that directly composing the fault model with the system model clutters the 'nominal' model with failure information, which obscures the nominal system functionality. This complexity may make model evolution difficult, error prone, and costly. In our opinion, the system model and the fault model should be defined separately and some automatic composition mechanism should be created allowing the system model and fault model to be easily merged for analysis. Third, although we were able to successfully analyze a realistic example, there are serious questions about the scalability of the analysis tools.

Acknowledgements

We would like to thank Dr. Steven Miller and Dr. Michael Whalen of Rockwell Collins Inc. for their valuable insights and feedback in course of our collaboration with them on this project.

References

1. SAE ARP4761. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. SAE International, December 1996.
2. Pierre Bieber, Charles Castel, and Christel Seguin. Combination of fault tree analysis and model checking for safety assessment of complex system. In In Proceedings of the 4th European Dependable Computing Conference on Dependable Computing, pages 19–31. Springer-Verlag, 2002.
3. M. Bozzano, A. Villafiorita, O. Kerlund, P. Bieber, C. Bougnol, E. Bde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Ldtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, and G. Zacco. Esacs: an integrated methodology for design and safety analysis of complex systems. In In Proceedings of ESREL 2003, pages 237–245. Balkema Publishers, June 15–18 2003.

4. Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: the fsap / nusmv-sa safety analysis platform. In In Proceedings of SAFECOMP 2003, pages 49–62, Edinburgh, 2003. Springer.
5. James Dabney and Thomas Harmon. Mastering Simulink. Prentice Hall, Upper Saddle River, NJ, 2004.
6. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. Proceedings of the IEEE, 79(9):1305–1320, September 1991.
7. Yiannis Papadopoulos and Matthias Maruhn. Model-based synthesis of fault trees from matlab-simulink models. In The International Conference on Dependable Systems and Networks (DSN'01), July 01–04 2001.
8. Yiannis Papadopoulos and John A. McDermid. Hierarchically performed hazard origin and propagation studies. In In Proceedings of the 18th International Conference, SAFECOMP'99, volume LNCS 1698. Springer-Verlag, 1999.
9. Esterel Technologies. Scade suite product description. http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html.
10. Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

Using Safety Critical Artificial Neural Networks in Gas Turbine Aero-Engine Control

Zeshan Kurd and Tim P. Kelly

High Integrity Systems Engineering Group,
Department of Computer Science,
University of York, York, YO10 5DD, UK
{zeshan.kurd, tim.kelly}@cs.york.ac.uk

Abstract. 'Safety Critical Artificial Neural Networks' (SCANNs) have been previously defined to perform nonlinear function approximation and learning. SCANN exploits safety constraints to ensure identified failure modes are mitigated for highly-dependable roles. It represents both qualitative and quantitative knowledge using fuzzy rules and is described as a 'hybrid' neural network. The 'Safety Lifecycle for Artificial Neural Networks' (SLANN) has also previously defined the appropriate development and safety analysis tasks for these 'hybrid' neural networks. This paper examines the practicalities of using the SCANN and SLANN for Gas Turbine Aero-Engine control. The solution facilitates adaptation to a changing environment such as engine degradation and offers extra cost efficiency over conventional approaches. A walkthrough of the SLANN is presented demonstrating the interrelationship of development and safety processes enabling product-based safety arguments. Results illustrating the benefits and safety of the SCANN in a Gas Turbine Engine Model are provided using the SCANN simulation tool.

1 Introduction

The application of Artificial Neural Networks (ANNs) within safety critical systems is highly desirable. One notable benefit includes the ability to learn and adapt to a changing environment. Another advantage is the ability to generalise outputs given novel data. The operational performance of ANNs can also exceed conventional methods in areas of pattern recognition and function approximation. These qualities enable applications to provide improved efficiency (in terms of reduced cost) and maximisation of performance in a changing operating context. Previous work has defined the "Safety Critical Artificial Neural Network" (SCANN) [1, 2]. One major benefit of the SCANN is that it provides a white-box view for its behaviour. It is a 'hybrid' system that exploits both fuzzy and neural network paradigms for mutual benefit and overcomes many of the problems identified for ANNs [3]. Behaviour is described qualitatively using fuzzy rules whilst neural network learning algorithms are exploited to manipulate the quantitative representation. Through the use of safety constraints (and constrained learning) the behaviour of the SCANN can be guaranteed to not lead to identified failure modes [1, 2] typically associated with control problems.

A “Safety Lifecycle for Artificial Neural Networks” (SLANN) has also been previously defined [4]. The SLANN encapsulates the main development tasks involved in developing ‘hybrid’ ANNs. Also included are suitable processes that aim to determine safety requirements and systematically deal with partial prior knowledge. Throughout the lifecycle, safety requirements are determined, faults identified and mitigated. The SLANN directly interfaces with the problem environment to capture an intentionally complete specification during design.

Both SLANN and SCANN offer the possibility of using neural networks and fuzzy logical systems in highly dependable roles within safety critical systems. The Rolls Royce Spey Gas Turbine Aero-Engine (GTE) has been chosen as a real world problem to evaluate the practicality of both SLANN and SCANN. In particular, tradeoffs between safety and performance are examined to assess the degree of SCANN performance impact (or advantage) in the presence of safety constraints.

The motivation for using the GTE is the potential for improved performance for situations such as engine degradation. There is also the prospect to improve hazard detection through health monitoring - ultimately leading to enhanced efficiency.

Section 2 describes the mechanism and schematic for the GTE. Section 3 demonstrates effectiveness by presenting a stepwise walkthrough of the SLANN to generate a SCANN. The potential of SCANN learning in the GTE is examined in section 4.

2 Gas Turbine Engine Mechanism

Gas Turbine Engines (GTE) are internal combustion heat engines which convert heat energy into mechanical energy. There are three main elements within the GTE namely; compressor, combustion chamber and a turbine placed on a common shaft.

The GTE illustrated in Fig. 1 describes the typical mechanism for producing thrust. The initial stage involves atmospheric air entering the engine body. Air then enters the compressor which is divided into the LP (Low Pressure) and HP (High Pressure) compressor units (twin-spool). Air pressure is first raised by the LP Compressor unit and then further increased by the HP Compressor unit. The Inlet Guide Vane (*IGV*) is used to match the air from the fan to the HP compressor characteristics. Pressurised air then reaches the combustion chamber where engine fuel is mixed with the compressed air and ignited at constant pressure. This results in a rise in temperature and expansion of the gases. A percentage of the airflow is then mixed with the combusted gas from the turbine exit. This is then ejected through the jet pipe and variable nozzle area to produce a propulsive thrust.

The GTE has been used for military and civilian applications and is known as an “air-breathing” engine since the engine airflow is 250 times its fuel flow rate. A ‘Spey’ GTE nonlinear thermodynamic model (Matlab & Simulink) was acquired from our sponsors QinetiQ. Table 1 describes the main inputs and outputs of the system (for open-loop and closed-loop control).

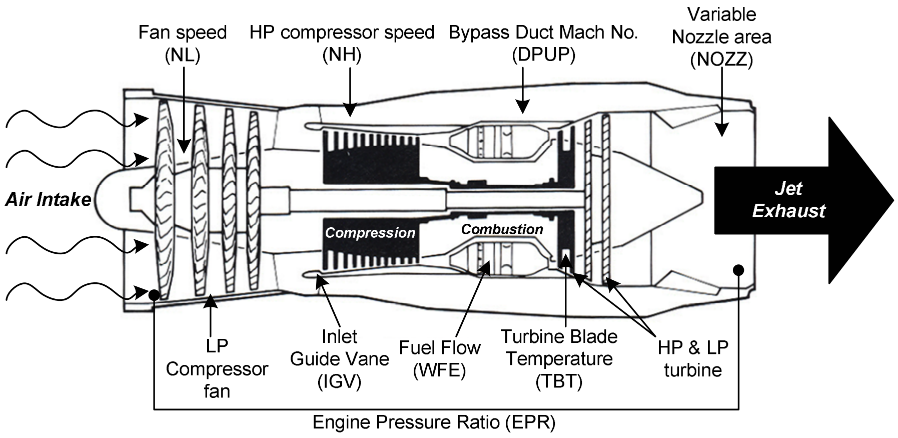


Fig. 1. Typical Twin-Spool Gas Turbine Aero-Engine Mechanical Layout

Previous work [5] has examined the potential to use fuzzy systems to replace several controllers in the GTE. This resulted in fuzzy schedulers for fuel flow, IGV and nozzle controllers using Mamdani and Takagi-Sugeno [6] fuzzy rules. The work demonstrated an unconventional approach for aerospace systems design to yield improved performance (such as thrust maximisation) over linear or non-linear polynomial schedulers. Although the study did make note of potential engine hazards the work focussed on performance issues instead of addressing hazards associated with each control function.

Table 1. Main inputs and outputs for the Simulink 'Spey' GTE model

Inputs of the GTE model		**Outputs of the GTE model**	
Variable	**Description**	**Vari-able**	**Description**
NHDem	Thrust setting and fuel flow demand (%). Rate limited to prevent over acceleration.	*NH*	High pressure spool speed (%). Air data is used to correct this value for changes in flight conditions.
WFE	Fuel Flow (kg/s)	*NL*	Low pressure spool speed (%)
HP IGV	Inlet Guide Vane (°)	*DPUP*	Bypass duct mach number
NOZZ	Exhaust nozzle area (m^2)	*XGN*	Gross thrust (kN)
		LPSM *HPSM*	Low/High Pressure Surge Margin measures how close engine is to stall. Indirectly controlled against DPUP and NL (%)
		TBT, JPT	Turbine and jet pipe temperatures (°C)

3 SCANN Development for the GTE IGV Scheduler Function

The SLANN defined in [1, 2] has been refined and presented in Fig. 2 with steps denoted by circled numbers. The development phases focuses on inserting prior knowledge (which may be partial and inaccurate) to generate a SCANN. It then goes through a process of learning until suitable performance is reached (and safety conditions satisfied). The refined knowledge is then inserted to generate a final SCANN. With further training (post-certification) this knowledge can be easily, completely and soundly extracted and analysed to learn about the problem domain [1]. Each step in the lifecycle is described and tackled in the following sub-sections.

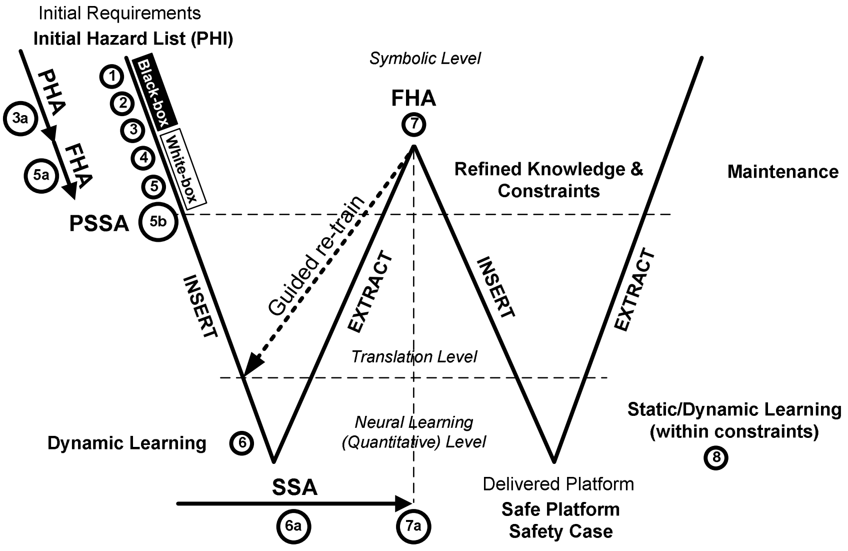


Fig. 2. Development and Safety Lifecycle for Artificial Neural Networks (SLANN) [4]

3.1 Problem Analysis

There are several components within the GTE control model that can be replaced with the SCANN including WFE, IGV, NOZZ and BOV (Blow-Off Valve). For the purpose of this paper, replacement of the conventional IGV scheduler is considered.

The role of the IGVs is to control airflow and maintain efficient fan operation. As the air passes from the trailing edge of the IGVs, the air drawn into the engine is deflected in the direction of the rotating compressor. The airflow angle of entry onto the rotating compressor blades must be within a stall-free range. This is achieved with a variable geometry IGV which changes the angle of attack of the blades to prevent compressor stall. Air pressure or velocity is not changed as a result of this action and is maintained within acceptable limits (for low airflow conditions). It also permits high airflow with minimum restrictions.

There are a substantial number of guide vanes within a compressor assembly. To allow variable guide vane positions, vane bearing seats are formed by radial holes and counterbores through circumferential supporting ribs. Typically, the positioning of the IGV angle cannot be achieved with a high degree of precision [5]. For control, the IGV is positioned against an open-loop mapping which is determined either by NH or NL as a SISO system (Single-Input Single-Output). The conventional IGV function is set to 32 degrees below 78% NH, 10 degrees above 91%, and with proportional operation between these ranges. This function ensures control between NH and NL and reduces risk of engine surge by maintaining a "working line". However, the linear schedule may not be optimal according to engine performance requirements [5]. The problem is to use the SCANN for more appropriate scheduling. This will replace the existing IGV scheduler whilst ensuring that identified hazards are mitigated and prevented.

3.2 Early Lifecycle Steps

To begin systematic development of the SCANN, **Step 1** of the SLANN involves the selection of input and output variables for the desired function. For the problem at hand, this task is simplified since a certified controller for the IGV already exists. Therefore the input for the IGV scheduler is NH with output IGV.

Step 2 is a design task which determines the appropriate reasoning mechanism. In this case, the mechanism will be the SCANN as defined in [1]. The SCANN is based upon the FSOM [7] and consists of six layers.

Layer 1 is the input layer and propagates inputs (from sensors) to layer 2. **Layer 2** is the fuzzy set membership (distance) function layer. This layer comprises of neurons for every input fuzzy set which perform the triangular function. **Layer 3** performs fuzzy inference (min or product operator) and has no adaptable parameters. **Layer 4** normalises activations (memberships) of activated rules. **Layer 5** computes crisp rule outputs using Takagi-Sugeno reasoning [6] described by (1).

$$y_i = f_i(x_1,\ldots,x_n) = a_{i,0} + a_{i,1}x_1 + a_{i,2}x_2 +,\ldots,+ a_{i,n}x_n. \tag{1}$$

Where y_i is the i^{th} rule output, $a_{i,n}$ are tunable parameters and x_n denotes the *n* input(s) into the SCANN.

Layer 6 consists of a solitary neuron whose purpose is to determine a single output value from several firing rules using weighted averaging (where weights are rule activations).

Step 3 determines the universe of discourse for each input and output variable. By examining the existing IGV function, the NH range is defined as [20, 115] and is rate limited.

Following this design task, Preliminary Hazard Identification (PHA) is performed (**Step 3a**) as an initial investigation of the potentially hazardous effects of control variable anomalies (expressed as failure modes). There is a HAZOP table for each input and output variable (Table 1) in order to determine the existence of potential system level hazards (using a black-box view). This provides a thorough approach for examining variables associated with the problem domain (such as addressing omission or commission of variables). In this case, variables NH and IGV have been clearly defined for existing schemes. Moreover, an argument can also be derived from this process about range and rate of change for each variable.

This analysis is similar to the FHA (Functional Hazard Analysis) performed for controllers in modern engines [8] and contributes to the identification of required safety constraints. Table 2 presents an extract from a PHA HAZOP table for IGV (output).

Table 2. Extract from a HAZOP table for the IGV output. The causes of each item include ice, wear or control. Remedies of each item are determined later.

No.	IGV	G. Word	Meaning	Consequence
1	Value	*MORE*	IGV value is too high	Stall/Surge/ Excess TBT/ Shaft Over-speed
2	Value	*LESS*	IGV value is too low	Stall/Surge/Excess TBT
3	Value	*NO*	Omission of IGV	Stall/Surge/Excess TBT
4	Value	*AS WELL AS*	Commission of IGV	Stall/Surge/Excess TBT
5	Value	*REVERSE*	Negative or positive	Stall/Surge/Excess TBT
6	Rate	*MORE*	Change is too high	Stall/Surge/Excess TBT
7	Rate	*LESS*	Change is too low	Stall/Surge/Excess TBT
8	Rate	*NO*	No rate change	Stall/Surge/Excess TBT
9	Rate	*REVERSE*	Change is decreasing or increasing	Stall/Surge/Excess TBT
10	Rate	*AS WELL AS*	Oscillations	Stall/Surge/Excess TBT

Although a similar table has also been generated for the input NH, constraining inputs lies outside the scope of the SCANN i.e. the SCANN output is 'safe' if the input is without hazards.

Potential hazards associated with the control of the IGV (described in Table 2) include engine surge – resulting in loss of thrust or engine destruction. This is caused by excessive aerodynamic pulsations transmitted throughout the whole engine (oscillations). For typical GTEs there is a surge line which is used as a measure of aerodynamic stability. This defines various surge points for different engine speeds. Another potential hazard is excessive Turbine blade temperature (TBT) leading to erosion of the turbine blades.

Another hazard is engine over speed. This is when the NH or NL shaft speeds exceed 101% which can lead to engine surge and other destructive problems. In modern engines there are various speed limiting features including Over Speed Governor units (OSG) and electronic control logic systems.

3.3 Steps 4 and 5: Fuzzy Variable Partitioning and Fuzzy Rule Formation

Step 4 is a design task which determines fuzzy sets qualitatively and quantitatively for the input space. Having a large number of fuzzy sets may improve the generalisation performance but reduce interpretability. Previous work on psychology and fuzzy systems design [9] find that the optimal number of fuzzy sets for each dimension is 7 ± 2. There are three main features of the IGV schedule and the fuzzy sets defined in Table 3 encapsulate the entire input range. To cater for partial knowledge, all desired fuzzy sets do not have to be defined at this stage (this will be resolved later). Any fault or uncovered regions in this partitioning will be discovered in the later steps of the SLANN.

Step 5 involves forming rules using the derived fuzzy sets. For SISO systems this is straight forward. If there are *i* fuzzy sets then *i* fuzzy rules are generated. In the case of MISO systems each fuzzy set in each input dimension are combined resulting in a rule for each possible combination. This provides input space coverage [1] or ε-completeness [10]. The initial set of rules for the IGV scheduler is shown in Table 3.

Table 3. Incomplete fuzzy rules for the IGV scheduler. Values in brackets define ranges for input and output sets.

	IF NH *is (Antecedent Fuzzy Set)*		**IGV** *is (Consequent)*	
Rule No.	**SET**	**SPREAD**	**SET**	**IGV**
1	LOW	$[sl_{Low}, sr_{Low}]$ = [19.9, 95]	VHIGH is [20,40]	32°
2	VHIGH	$[sl_{VHigh}, sr_{VHigh}]$ = [80, 115.1]	LOW is [0,10]	10°

Since it is difficult to assign a linguistic term to the bilinear consequent, a crisp value is used to describe a constant output function in Table 3 [11]. To enhance interpretability of the SCANN rules, the output space has also been partitioned. Therefore the consequent crisp value for each rule lies within a linguistically labelled set.

The quantification of the spreads and output do not have to be 'safe' at this design step and will be tackled later.

3.4 Step 5a: Functional Hazard Analysis (FHA)

Step 5a exploits a divide-and-conquer approach for identifying potential failure modes associated with each rule. HAZOP style guide words are applied to each rule generated (which may be partial and incorrect). The HAZOP in Table 2 is used to identify hazards associated with each rule and provide suitable remedies in the form of safety constraints. Each rule has its own HAZOP table – detailing functional level hazards rather than system level hazards. The result of this approach is shown in Table 4 which identifies failures modes of concern for a rule and the relevant safety requirements. A summary of failure modes which may occur is as follows:

- **Failure Modes 1 & 2:** IGV value is too high\low
- **Failure Modes 3 & 4:** IGV value omission\commission
- **Failure Modes 5 & 6:** IGV value change is increasing\decreasing
- **Failure Modes 7 & 8:** IGV value change is too high\low

Failure modes 1 and 2 relate to items 1, 2 and 5 in Table 2. The engine model tackles these failure modes by saturating the IGV output between 0 and 40 degrees. This is unacceptable for learning based systems since there are different bounds throughout the entire schedule (which can be dynamic). These can be prevented during normal

SCANN operation, learning and generalisation post-certification. Semantic constraints have been described in [2] which tackle this problem by providing preconditions and post-conditions (bounds) for each rule (contributing to a stability argument). A summary of semantic constraints are as follows:

1. Each fuzzy set spread edge is bounded between $[\min sl_{i,j}, \max sr_{i,j}]$
2. Rules outputs are bounded and saturated according to $[\min y_i, \max y_i]$
3. All rules have at least one overlapping rule
4. Rules with input set overlap must also have overlapping output bounds

Failure modes 3 and 4 (relating to items 3 & 4 in Table 2) can be mitigated by providing input space coverage:

1. Semantic constraints 1 to 4 above
2. All values in the defined input space must be a member of an input fuzzy set
3. Inputs beyond defined valid regions must not be members of fuzzy sets

Finally, failure modes 5 to 8 relate to items 6-10 in Table 2 and are tackled as follows:

1. Semantic constraints 1 to 4 above
2. The gradient of each rule output function must be constrained according to desired maximum and minimum output changes over input changes
3. Each rule must have at most one overlapping (input) rule and overlapping rules must have non-overlapping input regions
4. No subsumed rules must exist – those whose input preconditions are a subset of the preconditions of any other rule in the knowledge base
5. If a minimum rate of change is defined then the output for each rule must be within output bounds at each spread edge (no saturation)
6. For two rules *l* and *r* which overlap, the function y^* starting from y_r (at left side of overlap window) to y_l (at right side of overlap window) must abide by defined constraints
7. Instead of weighted averaging, the final output is y^* as described above

Safety constraints are exploited by the SCANN and the approach is more powerful and practical than safety 'monitors'. For example, conventional safety monitors make little provision for adaptive functions. Instead, the constraints are tightly integrated with the approximated function allowing learning post-certification.

Many of the engine parameters are cross-coupled. This means that any change in one will lead to disturbances with other control variables. The inherent non-linear dynamics and multi-variable nature presents an additional challenge to define suitable safety requirements for the IGV controller.

Obtaining safety requirements in Table 4 is realistic since they are typically available in modern engines and incorporated into validation schemes for control laws within FADECs (Full Authority Digital Electronic Control).

Indeed the initial rule base may be incomplete (prior knowledge). Nevertheless this safety process is performed at this stage as it helps provides an initial state for the next step and facilitates "design guidance".

Table 4. Extract from the FHA performed for each rule where remedies are obtained from example safety requirements

No	IGV Attrib.	Guide Word	Meaning	Cause	Consequence	Rem-edy
1	Value	*MORE*	Value is too high	Ice/Control/ Wear	Stall/Surge/E xcess TBT	32 upper bound
2	Value	*LESS*	Value is too low	Ice/Control/ Wear	Stall/Surge/E xcess TBT	25.3 lower bound
6	Rate	*MORE*	Output change is too high	Ice/Control/ Wear	Stall/Surge/E xcess TBT	Con-stant

3.5 Step 5b: Preliminary System Safety Assessment

Preliminary System Safety Assessment (PSSA) is concerned with determining whether the SCANN parameter state is faulty. The approach compares the actual SCANN state with the safety constraints described in step 5a. There are two possible results arising from PSSA. The first requires further training if any safety constraints are violated (safety-based stopping condition). The second result is that there are no violations (hence no systematic faults) for the defined safety requirements.

The dynamic learning phase in step 6 is flexible enough to not only mitigate faults through parameter tuning but also adapt the structure for new rules. If no further training is needed, it does not guarantee that the SCANN is 'safe'. This is because new rules may have been discovered which also need to be constrained. This is tackled by the iterative approach to discover new knowledge (step 6), apply constraints (step 7) and mitigate systematic faults (step 6 and 7a) with clear stopping conditions.

The SCANN simulation tool examined the state of the SCANN and determined if any conditions had been violated. Following the initial PSSA a number of faults were detected. Although a feature of the simulation includes the ability to automatically mitigate these faults, the identification of these faults was used to 'guide' the dynamic learning process (mitigation through training and assertion). The learning process can be focussed or directed to certain regions of the function. These areas may include unmapped regions of the input space, areas of identified faults, and rules where little is known about the outputs. Since the NH fuzzy set 'Medium' has been left undefined, the dynamic learning parameters will be set to add a set with large width. The maximum number of rules was set to 4 (to maintain interpretability).

3.6 Step 6: Dynamic Learning Phase

The aim of this step is two fold; to maximise generalisation performance and remove all systematic faults for the identified safety requirements. The SCANN\FSOM (Fuzzy Self-Organising Map) offers several learning algorithms [7] typically used in neural networks. Static learning algorithm uses a two phase approach for providing small changes to the SCANN:

- **Phase 1:** Parameters defining fuzzy set spreads (antecedents) are frozen and rule output parameters are tuned using the gradient descent algorithm [12]
- **Phase 2:** Consequent parameters are frozen and antecedent parameters tuned using a modified Least Vector Quantisation (LVQ) algorithm [7]

A particularly attractive feature of the SCANN is its ability to self-generate. This means it can choose to add new rules (decision based on heuristics) by generating appropriate neurons and links. This self-generating property is termed the "Dynamic Learning algorithm" and enables 'large' changes to the rule base. For this lifecycle step, unconstrained versions of learning algorithms are used. This is because learning may be necessary and usefully applied to adapt an initially unsafe parameter state to a state without constraint violations.

The training data consisted of 91 uniform samples which represent the standard non-optimal IGV scheduler described in section 3.1. The output functions were initially hand-tuned and learning was performed for 50 epochs using learning parameters which allow convergence. A performance-based stopping condition was defined and satisfied using Root Mean Square Error (RMSE).

After performing dynamic learning, remaining faults were identified using SSA (step 6a). No systematic faults remained for the defined safety requirements i.e. the dynamic learning phase safety-based stopping condition is satisfied. The state of the rule base after dynamic learning was extracted from the SCANN and is summarised in Table 5:

Table 5. New fuzzy rules (3 & 4) after dynamic learning for the IGV function

	IF *(Antecedent)*		*Consequent*
Rule No.	**NH**	**SET**	**IGV (d)**
1	LOW	[19.9,77]	32°
2	VHIGH	[90,115.1]	10°
3	MEDIUM	[60,80]	31°
4	HIGH	[75,92]	29°

3.7 Step 7: Functional Hazard Analysis

Following the dynamic learning process, FHA was applied to each of the extracted rules and result of this is shown in Table 6. Since new rules and safety constraints have been added, there is possibility of potential faults (safety constraint violations) which are tackled by the following step.

3.8 Step 7a: System Safety Assessment Revisited

An additional phase of SSA was performed to identify any safety constraint violations by the rule base as a result of additional safety requirements (items 3 and 4 in Table 6). In this case, additional systematic faults were identified and the process returned to step 6. After step 7a, assurance can be provided that the SCANN IGV function is in an initial safe state (and any future parameter states will also be safe).

Table 6. Summary of example safety requirements[1] for the complete knowledge

Rule No.	Semantic Constraints for NH	Semantic Constraints for IGV	Dir. of Change Constraints	Rate of Change Constraints
1	[19.9, 77]	[25.3, 32]	None	Constant
3	[70, 83]	[17,30]	Decreasing (-)	[1, 2]
4	[80, 91]	[9.87,20]	Decreasing (-)	[1, 2]
2	[88, 115.1]	[9.87,15]	None	Constant

4 Results of Learning and Generalisation Post Certification

Having developed a SCANN for IGV scheduling, the function (data set 1 in Table 7) learnt during the SLANN is without faults and all identified failure modes have been mitigated. Both the static and dynamic learning algorithms have been constrained to ensure that systematic faults are not incorporated (leading to failure modes) when adapting parameters. The constrained static learning algorithm (for on-line learning) is outlined below:

Constrained Static Learning Algorithm Outline

```
1. Let p(t) be the SCANN parameter state
2. Feed in training sample pair (desired input and output)
3. Perform centre, spread or output tuning using learning laws [7] on tp(t) – tem-
   porary copy of the SCANN parameter state which when tuned does not affect
   actual SCANN behaviour
4. Identify presence of any safety constraint violations (faults)
5. If no violation then use tp(t) for the new tuned SCANN state
       a. Otherwise, reject tp(t) and preserve training sample (reuse when
          learning rate is more decayed)
```

The dynamic learning algorithm needs to provide assurance that any new rule being added does not violate the current state of the SCANN. There are two approaches for adding new rules. The first is that any new rule which violates any of the safety constraints is not added. The second is that the algorithm analyses safety constraints of existing rules and inherits them for the new rule. Due to space constraints, the first approach is used for this example.

Table 7. Summary of training data used for the SCANN

Data Set	Samples	Distribution	Inferred function	Perf.	Convergence
1	91	Uniform	Safe	Sub-optimal	Yes
2	91	Uniform	Safe	Optimal	Yes
3	91	Arbitrary	Hazardous	Sub-optimal	Yes
4	91	Arbitrary	Hazardous	Sub-optimal	No (unstable)

[1] The actual values of these safety requirements are not representative of a real engine since they are based upon an engine model which features intentional inaccuracies.

SCANN learning can be exploited to maximise performance of the plant. In this case, a cost function is needed to generate suitable training samples and is difficult to determine due to cross-coupling of variables. For example, it has been identified that maximising thrust leads to degradation in other objectives (such as blade temperatures) [5]. Table 8 describes a set of safety criteria for the 'Spey' GTE which if violated would lead to the hazards outlined in Table 2.

The Multi-Objective Genetic Algorithm (MOGA) [5] has proved itself to be a versatile tool for finding optimal fuzzy schedulers for the GTE. The MOGA is composed of three levels and uses genetic algorithms to search for an optimal parameter state. The first two levels generate and analyse performance (using criteria in [5]) of potential solutions at different operation points (such as 54, 65, 75, 85, 95% NH). The last level selects the best fuzzy solution (by making trade-offs between objectives) to maximise thrust (XGN).

Although the MOGA algorithm could be used for the SCANN, hand-tuning of the IGV was performed to identify suitable cost function for SCANN learning (desired outputs for training samples). Training data acquired from the MOGA algorithm or other source can be of arbitrary integrity since the SCANN safety constraints provide assurance that hazards associated with the IGV schedule are prevented.

For the SCANN to learn the new IGV solution, static learning used data set 2 for 50 epochs. To analyse the effects of the SCANN on the GTE, safety criteria [5] in Table 8 was compared with the GTE state. The main requirements include preventing excessive TBT, avoiding surges and engine over-speed. This IGV solution (data set 2 in Table 7) satisfied the safety criteria whilst providing improved thrust (Table 8).

Table 8. Safety criteria typically used to determine hazards associated with the 'Spey' Engine obtained from [5] and the engine model. Results of the SCANN using four training sets are compared with the safety criteria for a non-degraded engine during worst-case NH change.

Description	Requirement	Data 1	Data 2	Data 3	Data 4
HPSM	$\geq 5\%$	$\geq 6.96\%$	$\geq 5.19\%$	$\geq 7.39\%$	$\geq 5.17\%$
LPSM	$\geq 5\%$	$\geq 10.6\%$	$\geq 9.5\%$	$\geq 10.51\%$	$\geq 9.48\%$
TBT	$\leq 1713°C$	$\leq 1558°C$	$\leq 1559°C$	$\leq 1544°C$	$\leq 1559°C$
NH Shaft Speed	$\leq 101\%$	$\leq 100.5\%$	$\leq 100.5\%$	$\leq 100.6\%$	$\leq 100.5\%$
NL Shaft Speed	$\leq 101\%$	$\leq 91.8\%$	$\leq 91.8\%$	$\leq 89.29\%$	$\leq 91.9\%$

The results show that the overall thrust has increased rising from an average of 20.1 *kN* (data set 1) to 21.3 *kN* (data set 2) leading to improved performance. On the other hand, the TBT has exhibited an overall increase temperature and the working line has been reduced as expected. Excessive NH and NL shaft speeds which are used to determine engine over speed are prevented during the course of the engine run (including worst-case of large acceleration followed by large deceleration). Table 8 also includes results of the GTE given arbitrary, hazardous training data which violated all safety requirements (data set 3). Data set 4 demonstrates ability of the SCANN to meet the criteria under unstable conditions by setting overly large learning rates. This was achieved by SCANN safety constraints which implicitly described conditions leading to the engine hazards. Results for data sets 3 and 4 (in Table 8) show that the safety criteria are met and Fig. 3 and Fig. 4 provide comparison of results for data sets 1 and 4.

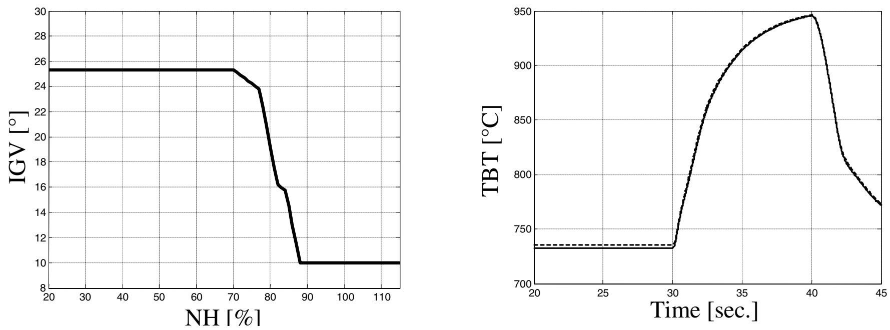


Fig. 3. Approximation of IGV schedule after learning training data 4 under unstable conditions *(left figure)*. Right figure is the results of Turbine Blade Temperature (TBT) for after learning data 1 *(solid line)* and 4 *(dotted line)*.

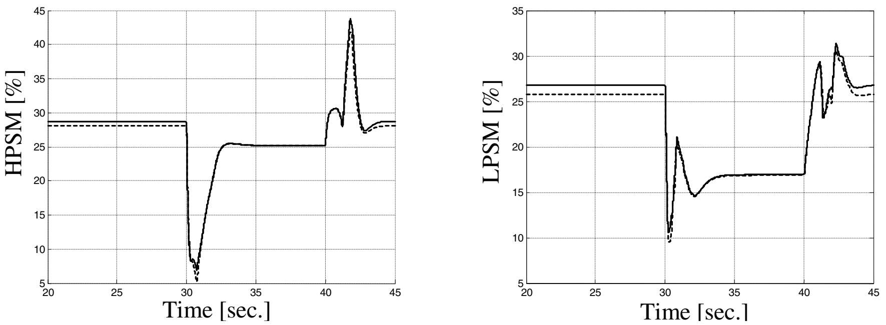


Fig. 4. HPSM and LPSM (over time) after SCANN learning data set 1 *(solid)* and 4 *(dotted)*

The SCANN can also contribute to health monitoring by making provision for hazardous operating contexts. Health monitoring is a technique for analysing degradation of a plant to determine whether servicing or maintenance is required. As a real-world example, the role of learning can include detecting actuator failure instead of thrust maximising (for advisory purposes). In the conventional control scheme, the IGV output is a *demand* which is fed into the actuators. Actuators then produce the *actual* IGV position value into the engine model. The inputs and actual IGV values can be paired and used as training data. The safety constraints can be exploited to provide additional information about plant performance. For example, continued semantic constraint violations may infer that either the training data is poor or the demands placed upon the scheduler have become hazardous (when actual moves too far from the desired IGV position). Logs of these attempted violations can be recorded and analysed to determine the state of the system as described in [1]. This provides additional exploitation of the SCANN learning algorithms for dynamic systems such as the GTE. This is an alternative approach for modern engine validation schemes which distinguishes itself by catering for the behaviour of intelligent adaptive systems.

5 Conclusions

This paper evaluates the practicality of using the SLANN and SCANN in a real-world problem. SCANN demonstrates the beneficial marriage of fuzzy logic systems and neural network paradigms. By exploiting a decompositional, analytical approach, the complete behaviour of the SCANN can be easily and soundly extracted and controlled through the use of safety constraints.

Feasibility and effectiveness of the safety and development processes in the SLANN has been demonstrated. Design phases use theoretical and empirical knowledge by directly interfacing and interacting with the environment.

The only major challenge within the SLANN is the activity of determining appropriate safety requirements. However, this task is no different than for conventional software safety or controller development.

The SCANN has established its ability to safely learn (post-certification) and thus improve GTE performance whilst maintaining safety requirements. Training data of arbitrary integrity is permitted (for post-certification learning) – allowing for more practical use of the SCANN in a real-world noisy environments. The advantages of generalisation and learning also include contributing to identifying plant degradation. Maintenance and servicing can also benefit (in terms of cost efficiency) through the use of logs generated during learning. This can contribute to maximising operating time for the plant before service is required leading cost efficiency. Both SLANN and SCANN enable product-based safety arguments to justify the use of neural networks and fuzzy logic systems in safety critical applications.

Acknowledgements

This work was funded by QinetiQ. The authors gratefully acknowledge the useful comments and engine model provided by S. Shackerly and S. MacKenzie both of QinetiQ (Farnborough).

References

1. Kurd, Z. and T.P. Kelly. Using Fuzzy Self-Organising Maps for Safety Critical Systems. in 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP'04), 21-24 September. Potsdam, Germany (2004)
2. Kurd, Z., T.P. Kelly, and J. Austin. Exploiting Safety Constraints in Fuzzy Self-Organising Maps for Safety Critical Applications. in 5th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL'04), 25-27 August. Exeter, UK (2004)

3. Kurd, Z., Artificial Neural Networks in Safety-critical Applications, First Year Dissertation, Department of Computer Science, University of York, 2002
4. Kurd, Z. and T.P. Kelly, Safety Lifecycle for Developing Safety-critical Artificial Neural Networks. 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP'03), 23-26 September, (2003).
5. Chipperfield, A.J., B. Bica, and P.J. Fleming, Fuzzy Scheduling Control of a Gas Turbine Aero-Engine: A Multiobjective Approach. IEEE Trans. on Indus. Elec. 49(3) (2002).
6. Sugeno, M. and H. Takagi. Derivation of Fuzzy Control Rules from Human Operator's Control Actions. in Proc. of the IFAC Symp. on Fuzzy Information, Knowledge Representation and Decision Analysis (1983)
7. Ojala, T., Neuro-Fuzzy Systems in Control, Masters Thesis, Department of Electrical Engineering, Tampere University of Technology, Tampere, 1994
8. Wilkinson, P. and T.P. Kelly. Functional Hazard Analysis for Highly Integrated Aerospace Systems. in IEE Seminar on Certification of Ground / Air Systems. London, UK (1998)
9. Miller, G.A., The magic number seven, plus or minus two: Some limits on our capacity for processing information. Psychol. Rev. 63(81-97) (1956).
10. Lee, C.-C., Fuzzy Logic in Control Systems: Fuzzy Logic Controller- Parts 1 & 2. IEEE Trans. on Systems, Man and Cybernetics. 20(2) (1990) 404-435.
11. Jang, J.S.R., ANFIS: adaptive-network-based fuzzy inference systems. IEEE Trans. Syst. Man. Cybern. 23(3) (1993) 665-685.
12. Haykin, S., Neural Networks: A Comprehensive Foundation: Prentice-Hall (1999).

On the Effectiveness of Run-Time Checks

Meine J.P. van der Meulen[1], Lorenzo Strigini[1], and Miguel A. Revilla[2]

[1] City University, Centre for Software Reliability, London, UK
`http://www.csr.city.ac.uk`
[2] University of Valladolid, Valladolid, Spain
`http://www.mac.cie.uva.es/∼revilla`

Abstract. Run-time checks are often assumed to be a cost-effective way of improving the dependability of software components, by checking required properties of their outputs and flagging an output as incorrect if it fails the check. However, evaluating how effective they are going to be in a future application is difficult, since the effectiveness of a check depends on the unknown faults of the program to which it is applied. A programming contest, providing thousands of programs written to the same specifications, gives us the opportunity to systematically test run-time checks to observe statistics of their effects on actual programs. In these examples, run-time checks turn out to be most effective for unreliable programs. For more reliable programs, the benefit is relatively low as compared to the gain that can be achieved by other (more expensive) measures, most notably multiple-version diversity.

1 Introduction

Run-time checks are often proposed as a means to improve the dependability of software components. They are seen as cheap compared to other means of increasing reliability by run-time redundancy, e.g. N-version programming.

Run-time checks (also called *executable assertions* and other names) can be based on various principles (see e.g. Lee and Anderson [3] for a summary), and have wide application. For instance, the concept of design by contract [5] enables a check on properties of program behaviour.

Some run-time checks can detect all failures, for example checks that perform an inverse operation on the result of a software component [1,2]. If the program computes $y = f(x)$, an error is detected if $x \neq f^{-1}(y)$. This is especially attractive when computing $f(x)$ is complex, and the computation of the inverse f^{-1} relatively simple. The argument is then that because computing f^{-1} is simple, the likelihood of failure of this run-time check is low. Also, it seems unlikely that both the primary computation and the run-time check would fail on the same invocation and in a consistent fashion. Together, these factors lead to a high degree of confidence that program outputs that pass the check will be correct. However—as these authors readily admit—such theoretically perfect checks do not exist in many cases, maybe even not in the majority of cases. Run-time checks can then still be applied, but they will in general not be capable of finding all failures. Examples of these partial run-time checks are given by e.g. [12].

Previous empirical evaluation of run-time checks have generally used small samples of programs, or single programs [4,7,11]. Importantly, we run these measures on a large *population* of programs. Indeed, if we wish to learn something general about a run-time check, we need this statistical approach. Measuring the effectiveness of a run-time check on a single program could, given a certain demand profile and enough testing, determine the fraction of failures that the check is able to detect (coverage) for that program, given that demand profile. But in practice, this kind of precise knowledge would be of little value: if one could afford the required amount of testing, at the end one would also know which bugs the program has, and thus could correct them instead of using the run-time check. However, a software designer wants to know whether a certain run-time check is worth the expense of writing and running it, without the benefit of such complete knowledge. The run-time check can detect certain failures caused by certain bugs: the coverage of the check depends on which faults the program contains; and the designer does not usually know this. What matters are the statistics of the check's coverage, given the statistics of the bugs that *may* be present in the program. If a perfect check cannot be had, a check that detects most of the failures caused by those bugs that are likely to be in a program has great value. A check that detects many failures that are possible but are not usually produced, because programmers do not make the mistakes that would cause them, is much less useful. In conclusion, the coverage of a check depends on the distribution of possible programs in which it is to be used.

Here, we choose three program specifications for which we have large numbers of programs, and for each of the three we choose a few run-time checks, then study their coverage. We thus intend to provide some example "data points" of how the coverage can vary between populations of programs. In addition to such anecdotal evidence—evidence that certain values or patterns of values *may* occur—such experiments may contribute to software engineering knowledge if they reveal either some behaviour that runs contrary to the common-sense expectations held about run-time checks, and/or some apparent common trend among these few cases, allowing us to conjecture general laws, to be tested by further research.

For lack of space, we only discuss coverage, or equivalently the probability of undetected failure. We will also not discuss other dependability issues like availability (possibly reduced by false alarms from run-time checks), although these should be taken into account when selecting fault tolerance mechanisms.

2 The Experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website [8] is an initiative of one of the authors (Revilla). It contains program specifications for which anyone may submit programs in C, C++, Java or Pascal intended to implement them. The correctness of a program is automatically judged by the "Online Judge". Most authors submit programs repeatedly until one is judged correct. Many thousands of authors contribute and together they have produced more than 3,000,000 programs for the approximately 1,500 specifications on the website.

Table 1. Some statistics on the three problems

	3n+1			Factovisors			Prime Time		
	C	C++	Pascal	C	C++	Pascal	C	C++	Pascal
Number of authors	5,897	6,097	1,581	212	582	71	467	884	183
First submission correct	2,479	2,434	593	112	294	41	345	636	125

We study the C, C++ and Pascal programs written to three different specifications (see Table 1 for some statistics, and `http://acm.uva.es/problemset/` for more details on the specifications). We submit every program to a test set, and compare the effectiveness of run-time checks in detecting their failures.

There are some obvious drawbacks from using these data as a source for scientific analysis. First, these are not "real" programs: they solve small, mostly mathematical, problems. Second, these programs are not written by professional programmers, but typically by students, which may affect the amount and kind of programming errors. We have to be careful not to overinterpret the results.

All three specifications specify programs that are memory-less (i.e. earlier demands should not influence program behaviour on later ones), and for which a demand consists of only two integer input values. Both restrictions are useful to keep these initial experiments simple and the computing time within reasonable bounds. The necessary preparatory calculations for the analysis of these programs took between a day and two weeks, depending on the specification.

2.2 Running the Programs

For a given specification, all programs were run on the same set of demands. Every program is restarted for every demand, to ensure the experiment is not influenced by history, e.g. when a program crashes for certain demands or leaves its internal state corrupted after execution of a demand (we accept the drawback of not detecting bugs with history-dependent behaviour). We set a time limit on the execution of each demand, and thus terminate programs that are very slow, stall, or crash. We only use the first program submitted by each author and discard all subsequent submissions by the same author. These subsequent submissions have shown to have comparable fault behaviour and this dependence between submissions would complicate any statistical analysis.

For each demand, the outputs generated by all the programs are compared. Programs that produce exactly the same outputs on every demands form an "equivalence class". We evaluate the performance of each run-time check for each equivalence class.

For all three specifications, we chose the equivalence class with the highest frequency as the *oracle*, i.e. the version whose answers we consider correct. We challenged each oracle in various ways, but never found any of them to have failed. For each specification, the test data were chosen to exhaustively cover a region in the demand space. In other words, we assume (arbitrarily) a demand profile in which all demands that occur are equiprobable.

2.3 Outcomes of Run-Time Checks

Run-time checks test properties of the output of a software component (the primary), based on knowledge of its functionality. In the rest of this paper we distinguish two types of run-time checks: *plausibility checks* and *self-consistency checks* (SCCs). The latter, inspired by Blum's "complex checkers" [12], use additional calls to the primary to validate its results, by checking whether some known mathematical relationship that must link its outputs on two or more demands does hold.

Checks on the values output by the primary are only meaningful if the output satisfies some minimal set of syntactic properties, one of which is that an output exists. Other required properties will be described with each specification. We call an output that satisfies this minimal set of properties "valid" (in principle this validaty check also constitutes a run-time check). We separate the check for "validity" from the "real" run-time checks, because it otherwise remains implicit and a fair comparison of run-time checks is not possible.

Table 2 shows how we classify the effects of plausibility checks. There are two steps: first, a check on the validity of the output of the primary; second, if this output is valid, a plausibility check on the output. There is an undetected failure (of the primary) if both the primary computes an incorrect valid output and the checker fails to detect the failure. Our plausibility checks did not cause any false alarms. Also note that a correct output cannot be invalid.

Table 2. Classification of execution results with plausibility checks

Output of primary	Output valid	Plausibility check	Effect from system viewpoint
Correct	Yes	Accept	Success
Correct	Yes	Reject	False alarm
Incorrect	Yes	Accept	Undetected failure
Incorrect	Yes	Reject	Detected failure
Incorrect	No	-	Detected failure

Table 3. Classification of execution results with self-consistency checks

Output of primary	Output valid	Output of second call to primary by self-consistency check	Effect from system viewpoint
Correct	Yes	Consistent	Success
Correct	Yes	Inconsistent	False alarm
Correct	Yes	Invalid output	Success
Incorrect	Yes	Consistent	Undetected failure
Incorrect	Yes	Inconsistent	Detected failure
Incorrect	Yes	Invalid output	Undetected failure
Incorrect	No	-	Detected failure

With self-consistency checks, the classification is slightly more complex (Table 3): we have to consider that one way the self-consistency check may fail is because its additional calls to the primary do not elicit valid outputs (e.g., they cause the primary to crash). We then assume that the self-consistency check will fail to reject the primary's output, i.e., that an undetected failure ensues. We could have made the decision to reject the output of the primary if the self-consistency check fails in this way; this would lead to slightly different results. False alarms did occur, which we do not analyse here for lack of space.

3 Results for the "3n+1" Specification

Short Specification. A number sequence is built as follows: start with a given number n; if it is odd, multiply by 3 and add 1; if it is even, divide by 2. The sequence length is the number of required steps to arrive at a result of 1. Determine the maximum sequence length (max) for all values of n between two given integers i, j, with $0 < i, j \leq 100,000$. The output of the program is the triple: i, j, max.

We tested "3n+1" with 2500 demands ($i, j \in 1..50$). The outputs of the programs were deemed correct if the first three numbers in the output exactly matched those of the oracle. We consider an output "valid" if it contains at least three numbers. In the experiment we discard non-numeric characters and the fourth and following numbers in the output. The programs submitted to "3n+1" have been analysed in detail in [9]; this paper provides a description of the faults present in the equivalence classes.

3.1 Plausibility Checks

We use the following plausibility checks for the "3n+1"-problem:

1. The maximum sequence length should be larger than 0.
2. The maximum possible sequence length (given the range of inputs) is 476.
3. The maximum sequence length should be larger than $log_2(max(i, j))$.
4. The first output should be equal to the first input.
5. The second output should be equal to the second input.

We measure the effectiveness of a run-time check as the improvement it produces on the average *probability of undetected failure on demand* (*pufd*). Without run-time checks, a program's probability of undetected failure equals its probability of failure per demand (*pfd*).

Figure 1 shows the improvement in average *pufd* given by these plausibility checks, depending on the average *pufd* of a pool of programs. We manipulate this average by removing, one by one, from the original pool of 13575 programs, the programs with the highest *pufd*. The more programs have been removed, the lower the average *pufd* of the remaining pool.

The graph clearly shows that many of these run-time check are very effective for unreliable programs (the right-hand side of the graph). More surprising is

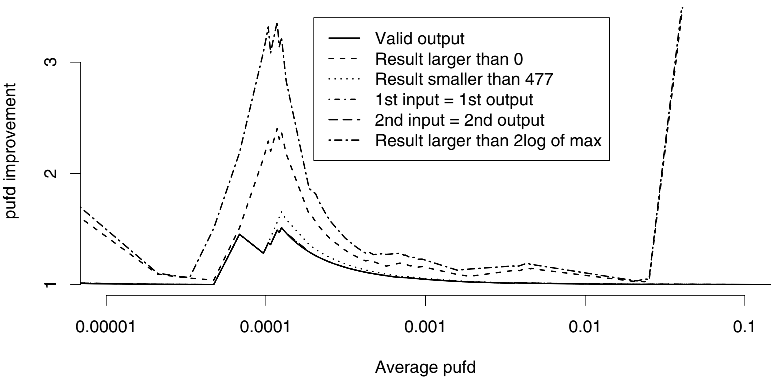


Fig. 1. The improvement of the *pufd* of the primary for the various plausibility checks for "3n+1". The curves for "1st input = 1st output" and "2nd input = 2nd output" are invisible because they coincide with the curve for "Valid output".

that the impact is quite pronounced at a *pufd* of the pool around 10^{-4}, while it is much lower for the rest of the graph. Apparently, these checks are effective for some equivalence classes that are dominant in the pool for that particular *pufd* range. Upon inspection, it appears that these programs fail for $i = j$.

The gain in *pufd* is for most of the graph only about 20%, but the peak reaches a factor of 3.2 for the plausibility check "Result $> log_2(max(i, j))$", a significant improvement over a program without checks. The check "Result>0" is mainly effective for programs that initialise the outcome of the calculation of the maximum sequence length to 0 or -1, if they abort the calculation before setting the result to a new value. This appears to be caused by an incorrect "for"-loop which fails when $i > j$. The check "Result<477" is not very effective. The failures it detects have mostly to do with integer overflow and uninitialised variables.

The check "Result $> log_2(max(i, j))$" is the most effective of all. It catches a few more programming faults than "Result > 0", especially of those programs that do not cover the entire range between the two inputs i and j for the calculation of the maximum sequence length.

Figure 2(a) gives some more detail of the performance of this plausibility check. It shows the percentage of failures detected for each equivalence class. We can make various observations. First, for many equivalence classes there is no effect (many crosses with a coverage of 0%). Second, since there are more crosses in the right-hand side of the graph, this check seems to be more effective when the primary programs tend to be less reliable (i.e., for development processes that tend to deliver poor reliability). We must say "seem" here, because this graph lacks information about the frequencies of the various programs (sizes of the equivalence classes). Third, this plausibility check still detects faults in the left-hand side of the graph, i.e. for the more reliable programs.

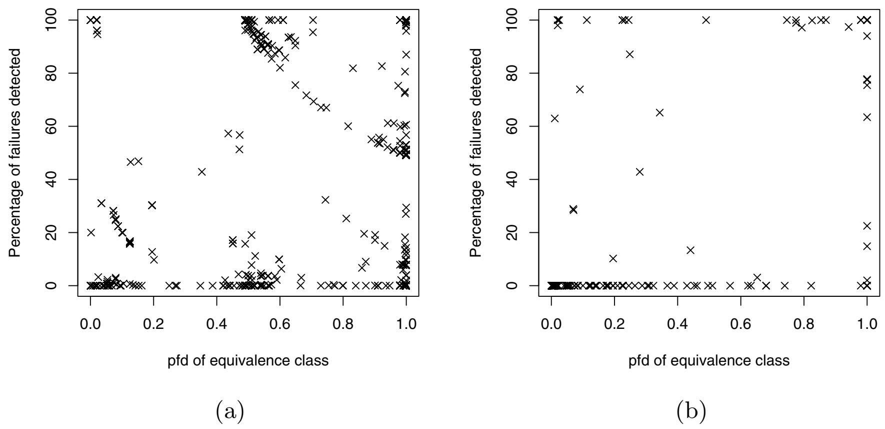


Fig. 2. Values of the error detection coverage of (a) the plausibility check "Result ¿ $log_2(max(i,j))$" for the equivalence classes of "3n+1" programs, and (b) the plausibility check "$i \leq j$" for the equivalence classes of "Factovisors" programs. Each cross represents an equivalence class. The horizontal axis gives the average *pfd* of the equivalence class, the vertical axis the percentage of its incorrect outputs that the check detects.

The plausibility check "First output equals first input" mainly catches problems caused by incorrect reading of the specification: some programs do not return the inputs, or not always in the correct order. These faults lead to very unreliable programs, and the effects of this plausibility check are not visible in Figure 1 because they manifest themselves (i.e. differ from the curve for "Valid output") for average *pufd*s larger than 0.1.

The result of the plausibility check "Second output equals second input" is almost equal to the previous one. There are a few exceptions, for example when the program returns the first input twice.

3.2 Self-consistency Checks

If we denote the calculation of the maximum sequence length as $f(i,j)$, then:

$$f(i,j) = f(j,i) \tag{1}$$

and:

$$f(i,j) = max(f(i,k), f(k,j)) \quad \text{for} \quad k \in i..j \tag{2}$$

and, if we combine these two properties:

$$f(i,j) = max(f(j,k), f(k,i)) \quad \text{for} \quad k \in i..j \tag{3}$$

Figure 3 presents the effectiveness of these self-consistency checks (for the experiment, we choose $k = \lfloor (i+j)/2 \rfloor$). Like our plausibility checks, these self-consistency checks appear to be very effective for unreliable programs.

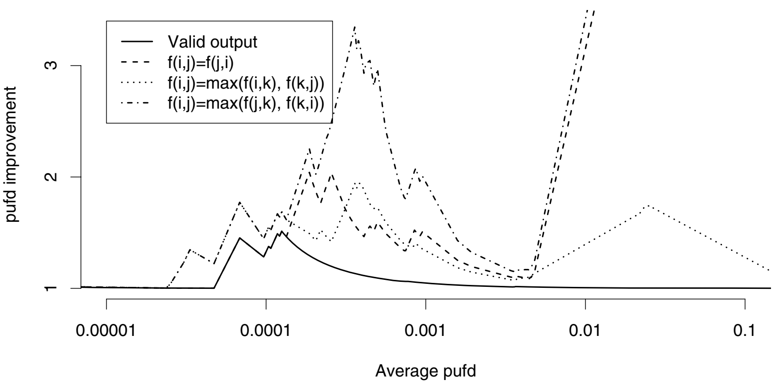


Fig. 3. Improvement in the average *pufd* of the primary for the various self-consistency checks for "3n+1"

The first self-consistency check mainly detects failures of programs in which the calculation of the maximum sequence length results in 0 or -1 for $i > j$. The second mainly finds failures caused by incorrect calculations of the maximum sequence length.

The third self-consistency check attains an improvement comparable to that of the plausibility check "Result $> log_2(max(i, j))$", but with a shifted peak. It appears that they catch different faults in the programs. As already stated, the peak of "Result $> log_2(max(i, j))$" is caused by programs failing for $i = j$ (which none of our self-consistency checks can detect) while this self-consistency check detects failures caused by faults in the calculation of the maximum sequence length as well as programs that systematically fail for $i > j$.

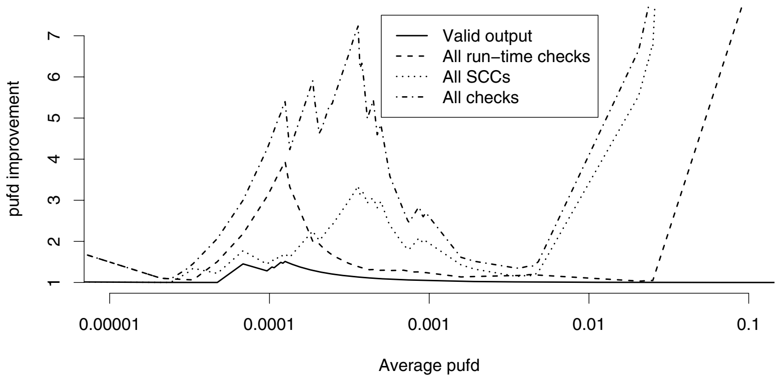


Fig. 4. Improvement in the average *pufd* of the primary for combinations of run-time checks for "3n+1"

The fact that the plausibility checks and the self-consistency checks tend to detect different faults is highlighted by Figure 4, which shows the performances of the combined plausibility checks, the combined self-consistency checks and the combination of all run-time checks.

4 Results for the "Factovisors" Specification

Short Specification. For two given integers $0 \leq i, j \leq 2^{31}$, determine whether j divides $i!$ (factorial i) and output "j divides i!" or "j does not divide i!".

We tested "Factovisors" with the 2500 demands ($i, j \in 1..50$). We consider an output "valid" if it contains at least two strings and the second is "does" or "divides". The main reason for invalid outputs appears to be absence of outputs.

4.1 Plausibility Checks

We use the following plausibility check for "Factovisors":

1. If $i \geq j$, the result should be "j divides $i!$".

Figure 2(b) shows the coverage of the run-time check "$i \geq j$" for each equivalence class. It is remarkable, again, that the crosses are spread over the entire plane: this check has some effect for equivalence classes with a large range of reliabilities. We also again observe the large number of crosses for a coverage of 0%, showing the check to detect no failure at all for that class of programs.

Figure 5 shows the *pufd* improvement caused by the plausibility check. As for "3n+1", we observe that the run-time check is very effective for unreliable programs. For pools of programs with average *pufd* between 10^{-4} and 10^{-2} the reliability improvement varies between 1 and 1.6.

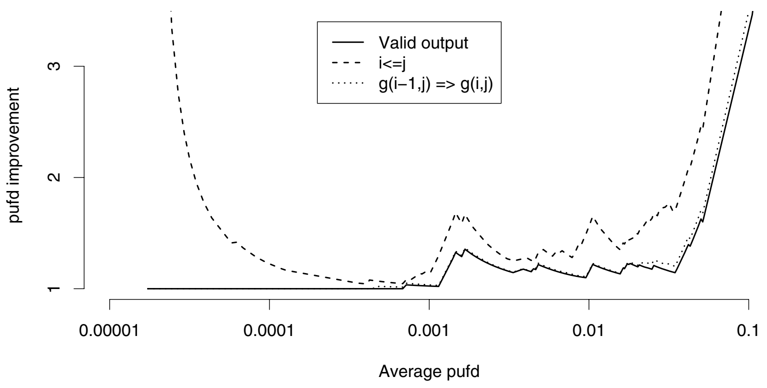


Fig. 5. The effectiveness of the run-time checks for "Factovisors"

The graph shows a peculiarity for *pufd*s smaller than 10^{-4}: the improvement approaches infinity. This is because as we remove programs from the pool, the faulty programs in the pool eventually become a "monoculture", a single equivalence class, and the check happens to detect all the failures of this class of incorrect programs. Here, the pool with the lowest non-zero average *pufd* contains 447 correct programs and 21 incorrect ones in the same equivalence class; the plausibility check detects the failures of these 21 incorrect programs.

4.2 Self-consistency Checks

If we call $g(i,j)$ the Boolean representation of the output of the program, with $g(i,j) = \mathit{true} \equiv$ "j divides i!", $g(i,j) = \mathit{false} \equiv$ "j does not divide i!", then:

$$g(i-1,j) \implies g(i,j) \quad \text{with} \quad i \neq 1 \tag{4}$$

As can be seen in Figure 5, the effect of this self-consistency check is minimal: the reliability improvement is never substantially greater than that given by the validity check.

5 Results for the "Prime Time" Specification

Short Specification. Euler discovered that the formula $n^2 + n + 41$ produces a prime for $0 \leq n \leq 40$; it does however not always produce a prime. Calculate the percentage of primes the formula generates for n between two integers i and j with $0 \leq i \leq j \leq 10{,}000$.

We tested "Prime Time" on 3240 demands ($i \in 0..79$, $j \in i..79$). The outputs were deemed correct if they differed by most 0.01 from the output of the oracle, allowing for round-off errors (the answer is to be given with two decimal digits).

The output is considered "valid" when it contains at least one number. We discard all non-numeric characters and subsequent digits from the output.

5.1 Plausibility Checks

The programs for "Prime Time" calculate a percentage, therefore:

1. The result should be larger than or equal to zero.
2. The result should be smaller than or equal to a hundred.

Figure 6 presents the effectiveness of the plausibility checks for "Prime Time". The plausibility check "Result $\geq$ 0" appears to have virtually no effect. The plausibility check "Result $\leq$ 100" has some effect, but not very large.

5.2 Self-consistency Checks

If we denote the result of the calculation of the percentage with $h(i,j)$, then:

$$h(i,j) = \frac{h(i,k) \times (k-i+1) + h(k+1,j) \times (j-k)}{j-i+1} \quad \text{for} \quad i \leq k < j \tag{5}$$

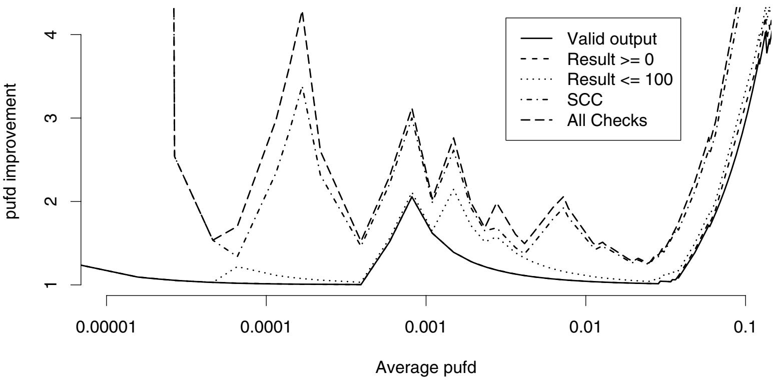


Fig. 6. The effectiveness of the run-time checks for "Prime Time". The curve for the plausibility check "Result $\geq$ 0" is not visible, because it coincides with the one for "Valid output".

Obviously, this check is not available when $i = j$. It is quite elegant: the computing time will not be excessively more than computing $h(i, j)$. For the experiment, we choose $k = \lfloor (i + j)/2 \rfloor$.

The effectiveness of the self-consistency check is shown in Figure 6. It is much more effective than the plausibility check "Result $\leq$ 100". We observe the same phenomenon for low *pufd*s as for "Factovisors": the effectiveness of the self-consistency check approaches infinity. When we combine the plausibility checks and the self-consistency check, we observe that the two complement each other: the combination is (slightly) more effective than the self-consistency check alone.

6 Run-Time Checks vs. Multiple-Version Diversity

A question that begs answering is: how do run-time checks compare to other forms of run-time fault tolerance? Using results we reported previously [10], we can compare our run-time checks against multiple-version diversity for "3n+1".

We observed (see Figure 7) that two-version diversity would become more effective with decreasing mean probability of failure on demand of the pool of programs from which the pair is selected, until a "plateau" is reached (between a *pufd* of 10^{-5} and 10^{-3}) with an improvement factor of about a hundred (note that the opposite trend—effectiveness decreasing with decreasing mean *pfd*—is also possible, as proved by models and empirical results [6]). For run-time checks the opposite occurs: their effectiveness decreases with decreasing average *pufd* of the primary reaching a fairly low improvement factor. The improvement factor of using diversity is significantly higher than that of applying run-time checks.

For these programs, it seems that these run-time checks could be the better choice for testing in the early phases of development, when the *pufd* of programs is still high, and multiple-version diversity when *pufd*s of programs become low.

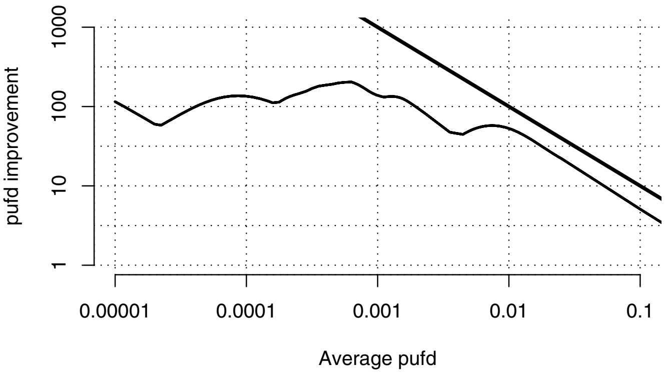


Fig. 7. Improvement of the *pufd* of a pair of randomly chosen C programs for “3n+1”, relative to a single version. The horizontal axis shows the average *pufd* of the pool from which both C programs are selected. The vertical axis shows the *pufd* improvement ($pufd_A/pufd_{AB}$). The diagonal represents the theoretical reliability improvement if the programs fail independently, i.e. $pufd_{AB} = pufd_A.pufd_B$. (This figure is based on [10].)

7 Conclusion

The results in this paper are of course specific to these three specifications, the programs submitted by these anonymous authors, the run-time checks we devised, and the demand profiles we used (uniform in a subset of the demand space). There are however some commonalities among the three sets of results, and we will tentatively discuss these here, while keeping in mind the limitations of this research.

First, we observe that the majority of the run-time checks considered are very effective for unreliable programs or have no effect at all.

Then, if we only look at pools of primaries with average *pufd* between 10^{-4} and 10^{-2}, the *pufd* improvement factor of the primary-checker pair is comparable for all three specifications: in the range 1–4. Over this range, the average improvement is less than 2 for all run-time checks considered.

Some run-time checks provide almost no benefit. It would be of great importance to be able to predict which checks are effective and which are not, but for the time being this seems not to be possible.

These plausibility checks appear to detect a different set of failures than the self-consistency checks, so that combining them is more effective than applying either one alone. So, the apparent “diversity” between the two kinds of checks did bring the benefit of some complementarity.

For pools of primaries with an average *pufd* lower than 10^{-2}, the *pufd* improvement achieved by the run-time checks considered for “3n+1” is far less than would have been achieved by applying multiple-version redundancy. In these analyses, the *pufd* improvement realised by multiple-version redundancy is at least a factor of a hundred better.

A natural comment on this work could be that since we have implemented simple-minded checks, it is not surprising that they only catch the simple-minded programming errors that cause highly unreliable programs. But this is actually a *non sequitur.* It is true that we do not expect expert programmers to produce highly unreliable programs, but our checks are "simple-minded" only in being based on simple mathematical properties of these specifications. There is no a priori reason why they should only catch simple-minded implementation errors: implementation errors are often caused by misunderstanding details of the specification or of the program itself, not of some mathematical property of the specification that is of little interest to the programmers. Likewise, there is no a priori reason for naive errors normally to cause faults which cause very high failure rates.

A tempting conjecture generalising the results we observed is that for some reason simple run-time checks tend (in some types of programs?) only to detect the failures in very unreliable programs. This would be an attractive "natural law" to believe and would simplify many decisions on applying run-time checks, so that it may be worth exploring further, since without some solid, plausible explanation (e.g. based on the psychology of programmers) or overwhelming empirical evidence, it would appear wholly unjustified.

Our measure of effectiveness as average improvement in *pufd* may be questioned. It is such that even if a check C has 100% coverage for the failures produced by a set of dangerous possible bugs, B, it will still be assessed as having negligible effectiveness if the bugs in set B occur with negligible probability in actual software development. Some may object that if C is the only check that can detect the effects of B-type bugs, and given the uncertainty on the probabilities of B these bugs being actually produced, a prudent designer will still use C. This objection is certainly right if C has negligible cost (implementation cost, cost in run-time resources, risk of bugs in C causing false alarms, etc). But whenever these costs are non-negligible, they must be weighted against C's potential benefits, as we do.

Acknowledgement

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

References

1. M. Blum and H. Wasserman. Software reliability via run-time result-checking. Technical Report TR-94-053, International Computer Science Institute, October 1994.
2. A. Jhumka, F.C. Gärtner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, École Polytechnique Fédérale de Lausanne (EPFDL), School of Computer and Communication Sciences, September 2002.

3. P.A. Lee and T. Anderson. *Fault Tolerance; Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer, 2nd edition, 1981.
4. N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self checks and voting in software error detection: An empirical study. In *IEEE Transactions on Software Engineering*, volume 16(4), pages 432–443, 1990.
5. B. Meyer. Design by contract. *Computer (IEEE)*, 25(10):40–51, October 1992.
6. P. Popov and L. Strigini. The reliability of diverse systems: A contribution using modelling of the fault creation process. *DSN 2001, International Conference on Dependable Systems and Networks, Göteborg, Sweden*, July 2001.
7. M. Rela, H. Madeira, and J.G. Silva. Experimental evaluation of the fail-silent behavior of programs with consistency checks. In *FTCS-26, Sendai, Japan*, pages 394–403, 1996.
8. S. Skiena and M. Revilla. *Programming Challenges*. Springer Verlag, March 2003.
9. M.J.P. van der Meulen, P.G. Bishop, and M. Revilla. An exploration of software faults and failure behaviour in a large population of programs. In *The 15th IEEE International Symposium of Software Reliability Engineering, 2–5 November 2004, St. Malo, France*, pages 101–12, 2004.
10. M.J.P. van der Meulen and M. Revilla. The effectiveness of choice of programming language as a diversity seeking decision. In *EDCC-5, Fifth European Dependable Computing Conference, Budapest, Hungary, 20–22 April, 2005*, April 2005.
11. J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *DSN 2001, International Conference on Dependable Systems and Networks, Goteborg, Sweden*, 2001.
12. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

A Technique for Fault Tolerance Assessment of COTS Based Systems

Ruben Alexandersson, Krishna Chaitanya D., Peter Öhman, and Yasir Siraj

Dept. of Computer Engineering,
Chalmers University of Technology, SE-41296, Gothenburg, Sweden
Telephone: +46(0)31-7721685
Fax: +46(0)31-7723663
ruben@ce.chalmers.se

Abstract. This paper investigates the feasibility of emulating source code software faults directly in Java byte code. Experimental results show that software defects introduced in source code can be emulated in Java byte code with a high level of confidence. This makes it possible to validate the dependability of Java programs with respect to realistic software defects embedded within the COTS components used without the need to know the source code. It is first investigated with good results how well the fault locations found at the byte code level map to the source code. The behaviors of the byte code level mutants are then compared with the corresponding source code mutant behavior. In a back-to-back comparative study with mutants based on ten representative programming defects, no difference in the program behavior between source and byte code level mutants could be distinguished.

1 Introduction

With software development proceeding at an unprecedented speed, in-house development of all system components will be too costly. Using commercial off-the-shelf (COTS) components reduces time-to-market, since the components are ready to be used, and saves money as the components are cheaper than developing from scratch. Quality and risk concerns currently limit the use of COTS components in safety and business critical applications. To increase the level of COTS usage in these application areas the techniques for dependability assessment of COTS-based systems must be further improved.

Software defects (also called faults or bugs) have been recognized as the major cause of computer outages [1]. It is however very difficult in practice to eliminate all software defects during development. Therefore, all non-trivial systems contain residual software defects that are activated when an appropriate input pattern is encountered during operation, which can lead to system failure and thus drastically affect system dependability.

Furthermore, as larger systems are more and more often being built with COTS components, the residual software faults embedded within the COTS represent a growing risk as the quality level of COTS components is difficult to assess [2]. A critical system built on COTS components should thus have robustness against faults embedded within these components. Since the source codes of COTS components are in most cases not available, it is not trivial to use fault injection techniques to experimentally verify the robustness of the system. Because of this, earlier efforts have focused on injecting errors at the interface between the COTS component and the rest of the system instead of injecting faults in the COTS, e.g. [3]. Although this is a useful technique, it can not be known how representative these errors are and whether they could actually have been generated by the activation of possible faults in the COTS. Thus, in order to generate error behavior that is representative of real software faults, we need a technique that can both determine the faults that could have been made within the COTS and inject them. We present a feasibility study of a mutation-based fault injection technique that shows great promise in achieving exactly that.

Fault injection is a well-known method for studying system behavior in the presence of faults. It is traditionally used to emulate external disturbances causing transient or permanent faults in the system under consideration. Residual faults like software defects, on the other hand, are most often not covered when experimental validation techniques such as fault injection are used. However, some studies, e.g. [4, 5, 6, and 7], have discussed the concept of injecting software faults for dependability evaluation. As residual software defects are difficult to activate (since they have passed the normal quality assurance process) they cannot be used to experimentally validate system dependability since the fault activation speed will be too low. Instead, representative software defects must be introduced into the system by fault injection in order to evaluate the capability of the system to cope with residual defects. This allows a significant speed up of fault activation.

The mutation technique was originally developed in the testing community as a means to assess test set quality. In fault tolerance evaluation, on the other hand, SWIFI has been the technique traditionally used for hardware related faults, e.g. [8]. This technique was extended by Costa and Madeira et al. [6, 7] to include software faults but was shown to have limitations. Because of this later work on software fault injection for fault tolerance evaluation purposes has taken up and used the technique of mutating program code [15].

Mutation-based fault injection of software defects is normally done by modifying elementary program components in the source code, which introduces small changes (faults) in the target program code, thus creating different versions of a program, and observing how each version behaves (each has one injected software fault) [9, 10]. The mutation testing community has long investigated the method of changing the source code. Modification of source code requires recompilation, re-linking and reloading, and this introduces a large overhead for fault injection experiments. Traditionally, this is somewhat enhanced by using interpretative tools. A method for avoiding this and speeding up the process is mutant schemata [11], but this technique is highly intrusive, and therefore in many cases not suited for fault tolerance assessment, and also still requires knowledge of the source code. Alternatively, modifying the machine code using low-level fault models would eliminate the time-consuming post injection process of compilation and linking without adding to the intrusiveness. This method is particularly useful when the source code is not available, which is the case with COTS components. Unfortunately this requires thorough analysis in order to be able to inject a set of low-level faults that corresponds to common high-level programming faults.

In a recent study, Madeira et al. [12] proposed a technique for emulating software faults through educated mutations introduced at the machine code level. The central idea is to find key programming structures at the machine code level where high-level software faults can be emulated. The results show that ODC classes [13] of faults, such as assignment, checking, interface and simple algorithm faults, can be directly emulated using this technique.

We extend these principles to be applied to the byte code for virtual machines such as JVM (Java Virtual Machine). Java byte code is the machine level representation of Java programs, just as real machine language is the representation of C or C++ programs. Since the Java byte code preserves the object structure of the source code we have the benefit of being able to emulate object-oriented faults, which is not possible in mutating real machine code. Java byte code is also closer in other aspects to the source code, with more instructions that uniquely map to specific source code constructs. We are interested in whether this gives us the possibility to mimic a larger set of the actual faults made at the source code level than is possible in real machine code. This could be highly usable and give reliable dependability evaluations.

The use of byte code level emulation was proposed by Ma et al. [10, 14] for mutant testing purposes. However, since their scope does not include the case in which the source code is unknown, they investigated a combination of mutant schemata and byte code translation for optimal performance.

In this paper we show that representative software defects in Java source code can be mapped to corresponding structures in the Java byte code. We can thereby emulate realistic software defects in components when the source code is unknown. Furthermore, this allows a significant speed up.

An alternative approach to the one presented in this paper would be to use an existing general purpose decompiler to get a source code representation of the COTS component and then apply source code based techniques for fault injection. However, a decompiler only needs to generate a valid source code representation of the byte code, not necessarily the one closest to the original. Hence there are questions regarding how representative the injected faults would be. A decompiler build for generating a source code as close to the original as possible have the exact same limitations as are meet when applying byte code level fault injection. Hence, the decompiler approach has no advantages compared to byte code level fault injection. Because of the significant performance benefit of injecting faults directly in the byte code this was the chosen approach.

2 Defect Mapping

2.1 Source Code Faults

The purpose of this mutation based method of fault injection is to make it possible to emulate the actual faults that are normally introduced into software during development and not like other methods, e.g. [3], to emulate the error behavior of such faults.

Fault injection experiments using actual software faults should give better confidence in the validity of the results obtained and should also be able to be used to verify the validity of error based methods. To define a fault model that can be considered representative of common software faults, that model must be based on field data from actual software projects. To the best of our knowledge no such survey of common faults has been published for the Java language. Duraes et al. [15] made a survey for the C language and, although it is likely that the fault types found for C will also to a large extent be representative of Java faults, this is not known. Because of this we can not base this study on a set of known common faults and verify that these specific faults can be emulated in Java byte code; instead we must evaluate which classes of faults, with respect to code constructs and structure, can be successfully emulated. The source code faults have thus been selected according to an emphasis on the criterion that they are suitable for testing the investigated method rather than their ability to stipulate a representative fault model for dependability evaluation. The criteria for selecting the faults have been:

- All applicable ODC classes [13] should be represented among the faults (i.e. Assignment, Checking, Interface and Algorithm)
- The set of faults should include the manipulation of variables, values, interfaces and program flow, spanning both classical procedural faults and object-oriented faults.
- The faults should differ in structure and the language constructs involved.
- The faults should be likely to occur from a programmer's mistake.

Using these criteria, a set of ten fault types that can be considered representative of the general case have been selected for the proof-of-concept experiments (see Table 1). Most of the fault types correspond to multiple source code patterns that are equivalent in complexity and structure. To speed up the experiment set-up phase, only a subset of patterns for each fault were considered in the experiments, which is adequate for showing feasibility. For example, only the byte code mapping (i.e. key programming structure) of two types of Wrong Logical Operator (WLO) faults, namely *`variable && variable`* and *`method-call && variable,`* were implemented. However, the WLO and WEB fault types have inherently significant differences in their respective set of source code patterns. Consequently, the results of these faults cannot be directly applied to the general case, but a more detailed investigation must be conducted (see section 4).

The detailed ODC classification suggested in [15] is used for the five classical procedural fault types (WVA, WLO, WEB, WPO and MBC). This detailed classification is not applicable for most of the object-oriented fault types, with the exception of the MOI fault, and therefore only the original ODC classification is used. As can be seen in Table 1, several of the fault types belong to more than one ODC class. This is due to the fact that the classification is determined by the context in which the fault appears. It can be noted that two of the selected faults match the Inter-Class operators of Ma et al. [17]. The DHV fault is equal to the IHI operator and the WOM fault is very similar to the IOD operator. The difference between WOM and IOD lies in that the mutation pattern of WOM changes the name of the method to simulate a typo and the IOD operator deletes the method altogether. Since the changed method name is never called by the surrounding program, the effect of the two is practically the same.

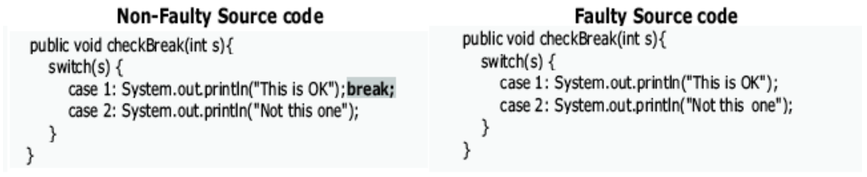
Table 1. The selected fault types

MOI – Missing Object Instantiation: The omission of instantiating an object variable when created. Can lead to failure if accessed prior to given a value	
ODC: Assignment (MVI)	*Limitation:* Only unparameterized instantiations are considered.
OCM - Object Changed by Method: An object given as input is altered by the method, although that was not intended.	
ODC: Assignment, Interface	*Limitation:* Only int variables within the object are altered.
WOM - Wrong Overriding Method name: A typo while writing an overriding method name will lead the compiler to treat it as a new one.	
ODC: Interface	*Limitation:* No limitation
WCT - Wrong Casting type: An object is being cast to a type that it does not have or is a subtype of.	
ODC: Assignment	*Limitation:* No limitation
DHV - Declaration of Hiding Variable: A variable is unintentionally given the same name as an instance variable in its own or an ancestor class, thus shadowing it.	
ODC: Assignment, Checking, Interface, Algorithm	*Limitation:* Only int variables that are accessed for reading are considered.
WVA - Wrong Variable Assignment: Variable is assigned to a wrong constant value.	
ODC: Assignment (WIDI)	*Limitation:* Only instance variables of type int are considered.
WLO - Wrong Logical Operator: Applying the wrong logical operator. For instance using || when it should be &&.	
ODC: Assignment (WLEA), Checking (WLEC), Interface (WLEP)	*Limitation:* Only `method && variable` and `variable && variable` expressions are considered.
WEB - Wrong Else Body: Omission of curly brackets surrounding a multi statement body of an `else` statement.	
ODC: Algorithm (MIEA), Checking (MIA)	*Limitation:* Only simple unnested if-else statements are considered.
WPO - Wrong Parameter Order: The order of equal type parameters are mixed up when making a method call.	
ODC: Interface (WPFO)	*Limitation:* Only int variables are considered.
MBC - Missing Break in Case: A break statement is unintentionally omitted at the end of a case body.	
ODC: Algorithm (MBC)	*Limitation:* No limitation

2.2 Fault Emulation Technique

The investigated technique is an adaptation of the G-SWFIT [12] method to Java programs. By introducing fault-specific changes directly into the byte code, the compiled result of a defect source code is emulated (without the need of an actual compilation). This requires knowledge of how Java source code is translated into byte code, in particular how high-level programming errors translate into specific instruction patterns at the byte code level.

The Missing Break in Case (MBC) fault is used as a running example throughout this section to explain the technique of finding (by the concept of key programming structures) and modeling (by fault injection patterns) a source code fault at the byte code level. A non-faulty and a faulty source code example are shown in Figure 1.


Non-Faulty Source code

```
public void checkBreak(int s){
   switch(s) {
      case 1: System.out.println("This is OK");break;
      case 2: System.out.println("Not this one");
   }
}
```

Faulty Source code

```
public void checkBreak(int s){
   switch(s) {
      case 1: System.out.println("This is OK");
      case 2: System.out.println("Not this one");
   }
}
```

Fig. 1. Non-faulty and faulty source code for the MBC fault

The part that requires the most effort is that of finding where the faults can actually be injected. As a first step, an environment for the fault at the source code level is defined. An environment is a specific set of source code statements where there is some room for programmers to make a mistake. For example, in the code given in Figure 2, a switch fall through occurs if the programmer forgets the break statement. Thus the Switch-Case statement as a whole forms the environment, thereby giving room for the programmer to introduce the fault, as shown in Figure 2.

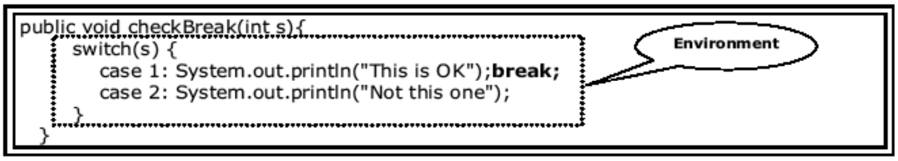
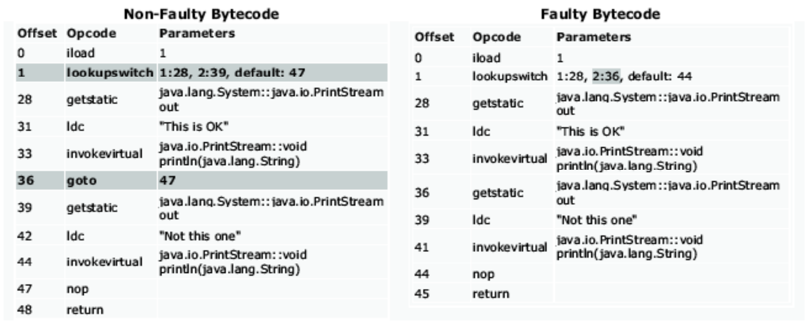

```
public void checkBreak(int s){
   switch(s) {
      case 1: System.out.println("This is OK");break;
      case 2: System.out.println("Not this one");
   }
}
```

Fig. 2. Environment of non-faulty source code

There can be more than one environment definition for a given fault type as there can be many such sequences/combinations/patterns that may lead to a fault of that particular type. For example, there are different environment definitions for the two types of WLO faults (see Table 1.) implemented.

Once the environment at the source code level is defined, the comparison of the code generated for both variants (with and without faults) allows us to identify the specific instruction patterns at the byte code level that are used to locate each fault (called the key programming structure) and the instruction patterns used to mutate the fault-free byte code (called the fault injection pattern).

This is illustrated in figure 3, which shows the set of correct and faulty byte codes for our running example. The key programming structure is highlighted in the non-faulty code to the left. The faulty byte code to the right is what is generated when the fault is made in the source code and then compiled. To mimic the behavior of the code to the right (and thereby emulate the fault) the parameter for the highlighted `goto` at offset 36 can be changed from 47 to 39. The fault injection pattern is therefore defined as making that change.



Non-Faulty Bytecode

Offset	Opcode	Parameters
0	iload	1
1	**lookupswitch**	**1:28, 2:39, default: 47**
28	getstatic	java.lang.System::java.io.PrintStream out
31	ldc	"This is OK"
33	invokevirtual	java.io.PrintStream::void println(java.lang.String)
36	**goto**	**47**
39	getstatic	java.lang.System::java.io.PrintStream out
42	ldc	"Not this one"
44	invokevirtual	java.io.PrintStream::void println(java.lang.String)
47	nop	
48	return	

Faulty Bytecode

Offset	Opcode	Parameters
0	iload	1
1	lookupswitch	1:28, 2:36, default: 44
28	getstatic	java.lang.System::java.io.PrintStream out
31	ldc	"This is OK"
33	invokevirtual	java.io.PrintStream::void println(java.lang.String)
36	getstatic	java.lang.System::java.io.PrintStream out
39	ldc	"Not this one"
41	invokevirtual	java.io.PrintStream::void println(java.lang.String)
44	nop	
45	return	

Fig. 3. Key programming structure at the byte code level

All standard source code compilers contain compile time checking mechanisms that verify certain aspects of a program, e.g. a variable is declared prior to being assigned a value. Therefore some faults made at source code level cannot pass the compilation and result in a faulty byte code executable. In conducting byte code fault emulation, these faults must be excluded by analyzing the mutant byte code.

In our running example with the switch fault we need to verify that the removal of the break statement does not lead to any unreachable statements that would have been recognized by the compiler.

The key programming structures and fault injection patterns for each fault type are the fundamental information needed to conduct the fault injections, as described in section 3.

3 Fault Injection and Evaluation

The fault injection technique is divided into two phases: first, the *fault analysis phase,* which searches for fault-related information in the byte code and marks the fault location, and, second, the *fault emulation phase*, which mutates the non-faulty byte code in a fault-specific way. The technique takes the non-faulty byte code as input and generates the mutated byte code that emulates the fault.

During the fault analysis phase the target application byte code is scanned for all the key programming structures defined for a fault. The search is based on a simple regular expression or a complex algorithm, depending on how the structure is defined. For each such hit, the corresponding fault location information is stored and passed on to the fault emulation phase.

During the fault emulation phase, the fault location information from the fault analysis phase is collected and mutated in a specific way defined by a fault injection pattern such that the resultant mutated byte code emulates the fault.

3.1 Prototype Description

A fault injection tool was developed for the proof-of-concept experiments using a pipe-and-filter architecture. This gives us the opportunity to collect and analyze the data output from the various components of the tool independently.

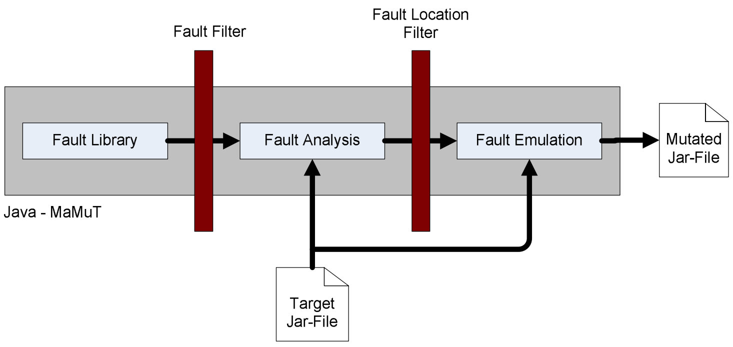


Fig. 4. Architecture of the fault injection tool

As can be seen in Figure 4, the tool consists of three main components. The fault library component is the container of the fault objects that holds information about the key programming structures and fault injection patterns for each fault type. The fault analysis component is responsible for analyzing the original byte code for faults and producing formatted fault location information (XML) by searching for possible fault locations in `.class` files (byte code) extracted from the jar file supplied. The fault emulation component performs fault-specific mutation on a given jar file according to the fault location information supplied by the analysis component and produces a mutated jar file as the result. Along with the mutant, it also produces a status report containing trace information about the mutations. The set of faults and fault locations used in the experiment can also be restricted with the aid of fault and fault location filters available in the tool.

3.2 Experimental Feasibility Evaluation

The experiment aims at determining the extent to which the byte code fault patterns emulate the corresponding source code defects. The fault injection process consists of both finding the correct locations (analysis phase) and manipulating the byte code (emulation phase). Consequently, the experiment is conducted in two parts. First it is investigated how well the fault locations found at the byte code level map to the source code. Next the behaviors of the byte code level mutants are compared with the corresponding source code mutant behavior.

Two target programs (named tp1 and tp2) are used in the experiment. The programs are selected on the criteria that it should be possible to manually find all possible source code fault locations and only a small number of test vectors should be needed for defect activation. The two target programs are thus fairly small (of a size of 20 and 70 KLOC) and sequential in structure so that the output is a strict function of the input. Both programs are implementations of a source code analysis tool and offer the same functionalities but employ different user interfaces (command line vs. graphical) and were implemented by different teams using different off-the-shelf parsers (with known source code).

3.2.1 Fault Location Experiment

As can be seen in Table 2, all the potential locations can be found, without any incorrect hits, for a vast majority of the fault types investigated. This means that it is possible by only analyzing the byte code to determine the actual number (and locations) of possible programming errors at the source code level of the investigated types.

Table 2. Byte code level fault location precision

Fault			Nr. of correct location identifications	Nr of incorrect locations	Nr. of missed locations	Ratio of correct locations
MOI	Missing Object Instantiation	tp1	73	0	0	100%
		tp2	213	0	0	100%
OCM	Object Changed by Method	tp1	34	0	0	100%
		tp2	439	0	0	100%
WOM	Wrong Overriding Method name	tp1	84	0	0	100%
		tp2	118	0	0	100%
WCT	Wrong Casting type	tp1	270	0	0	100%
		tp2	457	0	0	100%
DHV	Declaration of Hiding Variable	tp1	552	0	0	100%
		tp2	93	0	0	100%
WVA	Wrong Variable Assignment	tp1	15	0	0	100%
		tp2	49	0	0	100%
WLO	Wrong Logical Operator	tp1	1	0	0	100%
		tp2	5	0	0	100%
WEB	Wrong Else Body	tp1	4	0	0	100%
		tp2	22	0	0	100%
WPO	Wrong Parameter Order	tp1	21	0	0	100%
		tp2	33	0	0	100%
MBC	Missing Break in Case	tp1	306	20	38	84%
		tp2	1043	0	64	94%

As pointed out in section 2.1, some fault types correspond to multiple source code structures. For the WLO and WEB fault types, only the basic and least complex structure has been considered. That might indicate that we will have less optimistic data for the general case for these two fault types. A more comprehensive discussion about the generalization is given in section 4.

The missed and incorrect locations for the Missing Break in Case (MBC) fault type are the result of a fundamental difficulty with byte code level fault mapping. Many of the basic language constructs such as switch-case, if-else, loops and logical operations are translated to the same byte codes and can in some cases not be distinguished (see section 4 for a detailed discussion).

3.2.2 Fault Mutation Experiment

As Java programs are executed on a virtual machine there are inherent run time mechanisms for fault detection (i.e. exception handling). Successful fault detection by these mechanisms is manifested by an error signal such as a message on the stderr stream. In addition to this, application specific fault detection mechanisms (e.g. boundary checking of variable values) also use error signaling such as stderr or alert dialogue screens for fault detection signaling. When a fault is not detected an application can fail according to the well-known semantics of value and/or timing failures. Therefore it is natural to classify the application failure mode based on these three parameters (i.e. timing, output value and error signaling) as in Table 3.

Table 3. Application failure mode classification

Timing	Output	Error signal	Classificaton
OK	OK	No	Correct
OK	OK	Yes	Tolerated fault
OK	NOK	No	Undetected value failure
OK	NOK	Yes	Detected value failure
NOK	OK	No	Undetected timing failure
NOK	OK	Yes	Detected timing failure
NOK	NOK	No	Undetected arbitrary failure
NOK	NOK	Yes	Detected arbitrary failure

Since there is no time constraint for the target programs the only possible timing failure will be when the programs do not terminate (i.e. they “hang”).

The purpose of the experiment is to verify that the byte code mutants behave in the same way as the corresponding source code mutants. During the experiment, a detailed comparison was conducted between the output of the byte code mutants and source code mutants. The outputs of the mutated program were also evaluated by comparison with a reference of correct outputs and the corresponding application failure mode was determined. The correct locations found in the earlier experiment were used in this experiment. For eight of the ten fault types, all locations found were mutated. Since a very large number of locations were found for the WCT and MBC fault types, only a subset corresponding to the average number of mutants for the other fault types was included in the experiment, resulting in a total number of 2224 mutants.

Instead of using a large number of random test vectors, only one specifically selected for utilizing a large proportion of the program functionality was used. With this approach, a large number of fault activations were obtained with a minimum number of target program executions.

In a back-to-back comparative study with these 2224 mutants obtained by the ten fault types injected into the target programs, no difference in the program behavior between source and byte code level mutants could be distinguished. Not only did they show the same application failure profile but they also had the exact same outputs. This strongly indicates that the byte code level is feasible for emulating the erroneous behaviors of programs containing residual programming defects.

Table 4. Aggregated failure profile based on the proof-of-concept experiment

Fault		Correct	Tolerated Faults	Detected value failures	Undetected value failure	Timing failure (hang)	Total
MOI	Missing Object Instantiation	150	0	132	4	0	286
OCM	Object Changed by Method	469	0	3	1	0	473
WOM	Wrong Overriding Method name	196	0	3	3	0	202
WCT	Wrong Casting type	137	0	124	0	0	261
DHV	Declaration of Hiding Variable	612	0	26	7	0	645
WVA	Wrong Variable Assignment	48	0	11	5	0	64
WLO	Wrong Logical Operator	4	0	1	1	0	6
WEB	Wrong Else Body	26	0	0	0	0	26
WPO	Wrong Parameter Order	48	0	5	1	0	54
MBC	Missing Break in Case	144	0	62	1	0	207

As an example of the type of results that can be obtained by using the proposed method, Table 4 shows the failure mode classification for the experiment conducted. Since the emulation of source code level faults is perfect, identical application failure modes are obtained for both source and byte code level mutants. Hence, only one set of data is presented in the table.

The detection coverage for the value faults ranges between 50% and 100% depending on the fault type. This indicates that a large proportion of the faults are detected by exception handling mechanisms, as these were the only detection mechanisms available in the test programs.

The large number of correct outputs (i.e. the fault has not been activated or has been masked) owes mainly to the use of off-the-shelf parsers in the test programs as they contain a large proportion of unused functionality. A separate investigation of the experimental results showed that only 4% of the faults injected into the parser code led to a program failure whereas the corresponding figure for the application specific code was 53%. Fault activation is a well-known concern when conducting fault injection. The experiment conducted indicates that this problem is accentuated when using fault injection techniques on COTS- based software systems. However, static analysis of the target program code could determine which parts of the COTS components are never accessed and could be used to reduce the problem. Alternatively, a trace mutant could be used to dynamically determine which parts of the programs are activated by each test case.

4 Discussion

As seen in the experimental results the object oriented fault types (MOI, OCM, WOM, WCT and DHV) are successfully found and emulated at the byte code level. The basic (non-object oriented) faults fall into three categories. The first is associated with data or data containers (WVA) and the second concerns the manipulation of interfaces (WPO). Both containers and interfaces are very visible in Java byte code and the identification and modification of these have not presented any difficulties in this study.

As mentioned we implemented only a specific subset of source patterns for each fault in this experiment. For all fault types discussed so far the subset that we used is representative of the general case; consequently the method has a general feasibility for these fault types.

The case is somewhat more complicated for the third category of basic faults, namely the ones associated with program flow (i.e. WLO, WEB and MBC). The subsets that are implemented for two of these (WLO and WEB) are the basic and least complex ones, and thus the experimental results for these can not automatically be scaled to the general case. However, the problems associated with the general case were identified from implementing these subsets and from the full implementation of the Missing Break in Case (MBC) fault type. It should be noted that the problems discussed below concern local program flow (not passing any interfaces). If a fault stipulates mutation of program flow at the interface level it falls under the second category of basic faults and is easily achievable.

When conditional statements such as logical operations, if-else, switch-case and loops are translated into byte code they all get a similar structure that makes it very difficult, and in many cases probably impossible, to distinguish them from one another. The problem is further accentuated when these statements are nested together, which is very common in normal programs. This is a serious problem when working on real machine code and it seems to some extent to be a problem for Java byte code as well. However there are some constructs in Java byte code that we lack in real machine code that can be of aid in this process. As an example there is a unique code that is present whenever there is a switch statement. This is naturally helpful in distinguishing a switch-case from a series of if-else and it is the reason for the good results with the MBC fault type.

Thus, for this class of faults, Java byte code is still better suited for fault injection than real machine code, but it is uncertain whether it is sufficiently suited. A further and more detailed study of this class of faults is needed to fully understand the limitations of Java byte code in this regard.

5 Conclusions

In this paper we investigate a mutation-based fault injection technique that can be used for dependability evaluations of COTS-based systems. We show that representative software defects at the Java source code level can be mapped to corresponding structures at the byte code, thereby emulating realistic software defects in components where the source code is unavailable.

It is shown that byte code level mutants emulate source code level mutants for a set of ten representative programming source code defects with some minor restrictions in finding all fault locations in the byte code.

It is first investigated how well the fault locations found at the byte code level map to the source code. The study shows that a one-to-one mapping in fault location can be obtained for fault types related to object orientation, data/data container and interfaces. The program flow related fault types, on the other hand, are difficult to locate at the byte code level in some situations.

Second the behaviors of the byte code level mutants are compared with the corresponding source code mutant behavior. In a back-to-back comparative study no difference in program behavior between source and byte code level mutants could be distinguished. Not only did they show the same application failure profile but they also had the exact same outputs. This strongly indicates that the byte code is feasible for emulating the erroneous behaviors of programs containing residual programming defects.

The mutant-based fault injection technique investigated can therefore be used to validate the dependability of Java COTS-based systems with respect to realistic software defects without the need to know the source code. This also gives a significant speed up compared to source code based methods.

The results furthermore indicate that a large proportion of the faults is detected by exception handling mechanisms. We also show that the fault activation level is significantly lower in the off-the-shelf part than in the application specific part of the code.

Acknowledgements

This research was conducted within the CEDES project, which is funded by the Swedish industry and government joint research program IVSS - Intelligent Vehicle Safety Systems.

References

1. U. D. Commerce. The Economic Impacts of Inadequate Infrastructure for Software Testing. RTI, Research Triangle Park, NC 27709, US 2002.
2. S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. Metrics and models for cost and quality of component-based software. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 149-155, May 2003.

3. J. Voas, F.Charron, and K.Miller. Robust Software Interfaces: Can COTS-based Systems be Trusted Without Them?. In *Proceedings of the 15th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'96)*, October 1996. Springer-Verlag.
4. J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 304-313, June 1996.
5. E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, Volume 14(4), July-August 1997.
6. H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 417-426, June 2000.
7. Diamantino Costa, Tiago Rilho, M. Vieira and Henrique Madeira. ESFFI – A novel technique for the emulation of software faults in COTS components. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS)*, pages 197-204, April 2001.
8. J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: generic object-oriented fault injection tool. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN),* page 668, June 2003.
9. R. A. DeMillio, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Volume 11(4), pages 34-41, April 1978.
10. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. An Experimental Mutation System for java. *ACM SIGSOFT Software Engineering Notes*, Volume 29(5), pages 1-4, September 2004
11. R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software testing and Analysis*, July 1993.
12. J. Duraes and H. Madeira. Emulation of software faults by educated mutations at machine-code level. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 329-340, November 2002.
13. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong. Orthogonal defect classification - a concept for in-process measurements. *IEEE Transactions on Software Engineering*, Volume 18(11), pages 943-956, November 1992.
14. Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon. MuJava: An Automated Class Mutation System. In *The Journal of Software Testing, Verification, and Reliability*, Volume 15(2), June 2005
15. J. Duraes and H. Madeira. Definition of software fault emulation operators: a field data study. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN),* pages 105-114, June 2003.

Finding Upper Bounds for Software Failure Probabilities – Experiments and Results

Monica Kristiansen[1, 2]

[1]Østfold University College, NO-1751 Halden, Norwa
[2]Institute for Energy technology, NO-1751 Halden, Norwa
monica.kristiansen@hiof.no

Abstract. This paper looks into some aspects of using Bayesian hypothesis testing to find upper bounds for software failure probabilities, which consider prior information regarding the software component in addition to testing. The paper shows how different choices of prior probability distributions for a software component's failure probability influence the number of tests required to obtain adequate confidence in a software component. In addition, it evaluates different choices of prior probability distributions based on their relevance in a software context. The interpretations of the different prior distributions are emphasised. As a starting point, this paper concentrates on assessment of single software components, but the proposed approach will later be extended to assess systems consisting of multiple software components. Software components include both general in-house software components, as well as pre-developed software components (e.g. COTS, SOUP, etc).

1 Introduction

The use of software components in any kind of critical system requires evidence that the software component is dependable [13]. When focusing on reliability, which is one of the main attributes of dependability, this can be done by assessing the software component's failure probability and by demonstrating adequate confidence in this calculation.

In principle, a software component's failure probability can be assessed through statistical testing. However, since critical software components usually need to have low failure probabilities [12], the number of tests required to obtain adequate confidence in this failure probability often becomes practically very difficult to execute. An alternative approach is therefore to use all available prior information to compensate for the enormous number of tests required.

This paper studies the use of Bayesian theory [1, 4, 6, 18] and looks into Bayesian hypothesis testing [2, 19] as one possible approach to find upper bounds for failure probabilities in software components, which both takes prior information regarding the software component and testing into consideration. Different choices of prior probability distributions for a software component's failure probability are evaluated in a software context, and their influence on the number of tests required to obtain adequate confidence in a software component is investigated.

This paper concentrates on assessment of single software components, but the motivation for the proposed approach is a belief that it can be further extended to assess systems consisting of multiple software components [7, 9, 10]. Of special interest is the challenge to find upper bounds for simultaneous failure probabilities [3, 11, 12, 16]. Since the probability of simultaneous failures is likely to be significantly smaller than single failure probabilities, the number of tests required to obtain adequate confidence in these failure probabilities is practically impossible to execute. One of the main goals for further work is therefore to find upper bounds for simultaneous failure probabilities, and thus making it possible to include dependency aspects in software reliability models.

Software components include both general in-house software components, as well as pre-developed software components. Reusing pre-developed software components have become a common approach in software development due to the fact that this has the potential of significantly reducing development costs. Although reusing software components might benefit reliability as well as reducing costs, it will in many cases be difficult to assess whether the reliability is actually improved or not. Currently there is no broadly accepted way of including and assessing pre-developed software components in critical systems.

The rest of this paper is organized as follows. Chapter 2 presents some necessary notation, and describes the theory that forms the basis of the Bayesian hypothesis testing approach. In addition, some important definitions are included. In Chapter 3 the influence of different choices of prior probability distributions for a software component's failure probability is investigated. Chapter 4 discusses different choices of prior probability distributions for failure probabilities in a software context, and Chapter 5 concludes and describes further work.

2 Background

In this chapter a brief description of the theory that forms the basis of the Bayesian hypothesis testing approach is given. In addition some important definitions are outlined. Important notation used throughout this paper is listed in Table 1.

Table 1. Notation

Term	**Explanation**
n	= numbers of tests
r	= numbers of failures in n tests
θ	= a software component's failure probability
θ_0	= an accepted upper bound for a software component's failure probability
C_0	= a given predefined confidence level
$\pi(\theta)$	= prior distribution for a software component's failure probability (θ)
$\pi(\theta \vert D)$	= posterior distribution for a software components failure probability (θ)
$L(\theta \vert D)$	= likelihood function

2.1 Definitions

The terms *random failure* and *systematic failure* are often used for hardware failures and software failures, respectively [12], even though these terms are a bit misleading. Systematic failure refers to the fault mechanism where a fault reveals it self as a failure. So, if a software component failed once on a particular input it will always fail on that input until the fault has been successfully removed.

However, interest really centres upon the component's failure process: What happens when the component under study is used in its operational environment? Since there is uncertainty as to which input that will be selected on a particular occasion, there is uncertainty as to whether there will be a failure or not. In other words, a component's failure process is a stochastic process. Systematic failures, in the presence of non-deterministic usage, are therefore in reality just as random as random failures, and both random failures and systematic failures are susceptible to statistical analysis. Even though systematic failures usually are used for software failures, it should be emphasised that systematic failures also can arise from certain design and construction faults in hardware.

Statistical testing [5, 17] consists of exposing a piece of software to test cases drawn randomly according to some probability distribution defined over the program's input space. Such testing can be used to assess a software component's failure probability. Typical assumptions in statistical testing are (i) independent test runs, (ii) constant failure rate, (iii) all failures during testing are detected, and (iv) the operational profile is known.

One benefit of statistical testing is that it requires no knowledge of the internal structure of the software component being tested. This is of great benefit when pre-developed software components are used, for which one might not have all the required information available. Pre-developed software is in this paper defined as: "*Software which already exists, is available as commercial or proprietary product and is being considered for use in a computer-based system*" [8]. This definition encompasses any kind of reuse of software whether it is black box, commercially available, from an in-house library, or just happens to be available from another system.

Let θ_0 denote the accepted probability of failure for a given software component. The number of fault free tests, *n*, which must be carried out to satisfy the failure probability θ_0 and the given predefined confidence level C_0, using classical statistical testing, is given in equation (1), [15].

$$n = \frac{\ln(1-C_0)}{\ln(1-\theta_0)} \tag{1}$$

2.2 Bayesian Analysis

Bayesian analysis [1] consists of combining prior information ($\pi(\theta)$) and sample information (D) into a posterior distribution ($\pi(\theta|D)$) for θ given D. It is from this posterior distribution all decisions and inferences are made in Bayesian analysis. Bayes theorem [1] is expressed in equation (2), where the prior distribution $\pi(\theta)$

reflects beliefs about θ prior to testing, and the posterior distribution $\pi(\theta|D)$ reflects updated beliefs about θ after testing.

$$\pi(\theta \,|\, D) = \frac{L(\theta \,|\, D)\pi(\theta)}{\int_{\theta} L(\theta \,|\, D)\pi(\theta)d\theta} \qquad (2)$$

In hypothesis testing, a null hypothesis $H_0(\theta)$ and an alternative hypothesis $H_1(\theta)$ are specified. In classical statistics one decides between H_0 and H_1 by examining type I and type II error probabilities. These probabilities of error represent the chance that a sample is observed for which the test procedure will result in the wrong hypothesis being accepted. Type I error occurs when H_0 is rejected when it is true, and type II error occurs when H_0 is accepted when it is false.

In Bayesian analysis, hypothesis testing is conceptually more straightforward. One calculates the posterior probabilities $\alpha_0 = P(H_0|D)$ and $\alpha_1 = P(H_1|D)$, which combine both test data and prior knowledge, and decide between H_0 and H_1 accordingly [1]. Often it is convenient to summarize the evidence in term of posterior odds. Saying that $\alpha_0/\alpha_1 > R$, clearly says that H_0 is R times as likely to be true than H_1.

Although the posterior probabilities of the hypotheses are the primary measures in Bayesian hypothesis testing, the prior probabilities $\pi_0 = P(H_o)$ and $\pi_1 = P(H_1)$ are also of interest. π_0/π_1 is called the prior odds ratio, and the Bayes factor can be expressed by combining the posterior and prior odds ratios, as shown in equation (3).

$$B = \frac{\alpha_0 / \alpha_1}{\pi_0 / \pi_1} = \frac{\alpha_0 \pi_1}{\alpha_1 \pi_0} \qquad (3)$$

The Bayes factor can be viewed as a weighted likelihood ratio of H_0 to H_1 [1]. A Bayes factor greater than one indicates evidence in favour of the null hypothesis, and the higher the Bayes factor is the more evidence one has in favour of H_0. The Bayes factor forms the basis for finding the number of tests, required to satisfy a predefined upper bound θ_0 and confidence level C_0, in the proposed approach [2, 19]. This is outlined in more detail in the following chapters.

2.3 Finding Upper Bounds for Software Failure Probabilities by Using Bayesian Hypothesis Testing

One possible approach to find upper bounds for software failure probabilities is to use Bayesian hypothesis testing [2, 19]. Assume for further reading that H_0 and H_1 are specified as: H_0: $\theta \leq \theta_0$ and H_1: $\theta > \theta_0$, were θ_0 is a probability in the interval (0, 1). θ_0 represents an upper bound for a software component's failure probability and is assumed to be application specific and predefined (e.g. from standards, regulation authorities or customers).

In this case the null hypothesis states that the probability of software component failure is lower than an upper bound θ_0, whereas the alternative hypothesis states that the probability of software component failure is higher than an upper bound θ_0.

Often, it is convenient to express the prior belief in a software component's failure probability (θ) as probability distributions over the following two intervals $(0, \theta_0)$ and $(\theta_0, 1)$ [1]. This is shown in equation (4).

$$\pi(\theta) = \begin{cases} P(H_0)g_0(\theta), \theta \le \theta_0 \\ P(H_1)g_1(\theta), \theta > \theta_0 \end{cases} \tag{4}$$

The probability distributions; $g_0(\theta)$ and $g_1(\theta)$ describe how the prior mass is spread out over the two hypotheses. To reflect prior beliefs, the beta distribution is often chosen, since this distribution is a rich and tractable family that forms a conjugate family to the binominal distribution. The probability distribution for observing *r* failures during testing, given *n* independent trials and a constant failure probability (θ), is expressed by the binominal probability distribution. This is shown in equation (5).

$$f(r \mid \theta, n) = \binom{n}{r} \theta^r (1-\theta)^{n-r} \tag{5}$$

Based on equation (4) and (5), the posterior odds ratio can be expressed as shown in equation (6).

$$\frac{\alpha_0}{\alpha_1} = \frac{P(H_0 \mid r, n)}{P(H_1 \mid r, n)} = \frac{\int_0^{\theta_0} f(r \mid \theta, n) P(H_0) g_0(\theta) d\theta}{\int_{\theta_0}^{1} f(r \mid \theta, n) P(H_1) g_1(\theta) d\theta} \tag{6}$$

Further, it can easily be shown that the Bayes factor can be given as shown in equation (7).

$$B = \frac{\int_0^{\theta_0} f(r \mid \theta, n) g_0(\theta) d\theta}{\int_{\theta_0}^{1} f(r \mid \theta, n) g_1(\theta) d\theta} \tag{7}$$

For acceptance, the posterior probability for H_0 should be higher than a given predefined confidence level C_0. Based on this acceptance criterion, it can easily be shown that the Bayes factor also can be expressed as shown in equation (8).

$$B = \frac{C_0 P(H_1)}{(1-C_0)P(H_0)} \tag{8}$$

3 Experiments and Results

In this chapter, three different prior probability distributions for a software component's failure probability are investigated, and their influence on the number of tests required to obtain adequate confidence in a software component is studied.

The first case is based on earlier work done by Cukic et al. [2] and Smidts et al. [19], and assumes a uniform prior probability distribution both under the null hypothesis and the alternative hypothesis. Related to this prior probability distribution, there are some interesting aspects that need to be studied further. Two new cases are therefore used to investigate these aspects in Case 2 and Case3.

Case 1:
In papers [2, 19], two uniform probability distributions are used to describe how the prior mass is spread out over the two hypotheses. The prior information regarding θ is expressed in equation (9).

$$\pi(\theta) = \begin{cases} P(H_0)\dfrac{1}{\theta_0}, & \theta \le \theta_0 \\ P(H_1)\dfrac{1}{(1-\theta_0)}, & \theta > \theta_0 \end{cases} \tag{9}$$

In addition, it is assumed that no failures have been encountered during testing. Based on the binominal distribution in equation (5) and the prior belief about θ in equation (9), the Bayes factor, as defined in equation (7), can be calculated as shown in equation (10).

$$B = \frac{(1-\theta_0)(1-(1-\theta_0)^{n+1})}{\theta_0(1-\theta_0)^{n+1}} \tag{10}$$

The number of tests required to satisfy a given predefined confidence level C_0 can be found by combining equations (8) and (10), and is given in equation (11).

$$n = \frac{-\ln\left[\dfrac{C_0\theta_0 P(H_1)}{(1-C_0)(1-\theta_0)P(H_0)}+1\right]}{\ln(1-\theta_0)} - 1 \tag{11}$$

In the case where $P(H_0) = \theta_0$ and $P(H_1) = (1-\theta_0)$ the same result as by assuming an uniform prior probability distribution for θ over the entire interval (0, 1) is achieved and, as shown in equation (12), the required number of tests turns out to be almost the same as by using classical statistical testing.

$$n = \frac{\ln(1-C_0)}{\ln(1-\theta_0)} - 1 \tag{12}$$

Table 2 is extracted from [2] and shows the required number of tests for different choices of $P(H_0)$ and θ_0. The predefined confidence level C_0 is assumed to be constant equal to 0.99.

From Table 2 it can be seen that the number of tests, required to obtain adequate confidence in a software component, is greatly reduced when a uniform prior probability distribution is assumed both under H_0 and H_1. In addition, it can be seen that the higher the prior belief in H_0 is the fewer tests are needed. However, there are some interesting aspects related to this prior probability distribution:

Table 2. Required number of tests for different choices of $P(H_0)$ and θ_0, when a uniform prior probability distribution is assumed both under H_0 and H_1

θ_0	C_0	No. of tests $P(H_0) = 0.01$	No. of tests $P(H_0) = 0.1$	No. of tests $P(H_0) = 0.6$
0.01	0.99	458	228	50
0.001	0.99	2378	636	63
0.0001	0.99	6831	853	65
0.00001	0.99	9349	886	65
0.000001	0.99	9752	890	65

– First of all, a flat probability distribution for θ is assumed both under the null hypothesis and the alternative hypothesis (see Figure 1). This corresponds to the view that it is just as likely that the failure probability is close to 1 as it is close to θ_0 under the alternative hypothesis. In Case 2, this aspect is mitigated by allowing an expert to set a certain upper bound on the failure probability.

– Secondly, the probability distribution for θ is discontinuous in θ_0 (see Figure 1). This reflects a high prior belief that the failure probability is below θ_0, but it also opens for a small possibility that there is an unknown failure mechanism with a non-informative prior failure distribution. This aspect is mitigated in Case 3, by using a continuous beta distribution.

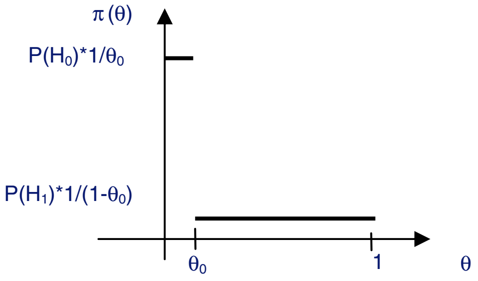


Fig. 1. The prior probability distribution used in Case 1

In addition it can be shown that it is the choice of two separate uniform probability distributions under the null hypothesis and the alternative hypothesis that results in the extremely low number of required tests. The uniform probability distribution on the interval (a, b) is expressed in equation (13).

$$P(\theta) = \begin{cases} \frac{1}{b-a}, & a < \theta < b \\ 0, & \text{elsewhere} \end{cases} \tag{13}$$

From this expression it can easily be seen that the smaller the denominator is, the higher the probabilities are. Since the area under the null hypothesis usually is extremely small compared to the area under the alternative hypothesis, the probability for failure probabilities less than θ_0 is much higher then the probability for failure probabilities higher than θ_0. This shows that the use of two separate uniform probability distributions under H_0 and H_1 not at all represents a conservative approach, even though the use of a uniform probability distribution over the entire interval usually is seen as an ignorance prior.

A somewhat worrying observation is that the number of tests, required to obtain adequate confidence in a software component, increases significantly when other more realistic distributions for a software component's failure probability are used. If an expert is allowed to set a certain upper bound on the failure probability or if a continuous probability distribution for the software component's failure probability is assumed, experiments show that the number of required tests increases significantly. This is outlined in more detail in Case 2 and in Case 3.

Case 2

One possible way to mitigate the effect of using a flat distribution under the alternative hypothesis is to allow an expert to set a certain upper bound on the failure probability under H_1, i.e. to state a value θ_1 for which the probability of having a failure probability higher than this is zero (see Figure 2). By assuming a uniform probability distribution both under H_0 and H_1, the prior probability distribution for θ can be expressed as shown in equation (14).

$$\pi(\theta) = \begin{cases} P(H_0)1/\theta_0, & \theta \leq \theta_0 \\ P(H_1)1/(\theta_1 - \theta_0), & \theta_0 < \theta >\leq \theta_0 \\ 0, & \theta > \theta_1 \end{cases} \tag{14}$$

As in Case 1, it is assumed that no failures have been encountered during testing. Based on the binominal distribution in equation (5) and the prior belief about θ in equation (14), the Bayes factor, as defined in equation (7), can be calculated as shown in equation (15).

$$B = \frac{(\theta_1 - \theta_0)(1-(1-\theta_0)^{n+1})}{\theta_0((1-\theta_0)^{n+1} - (1-\theta_1)^{n+1})} \tag{15}$$

By combining equation (8) and (15), the number of tests required to satisfy a given predefined confidence level C_0, can be found. This is expressed in equation (16).

$$\frac{1-(1-\theta_0)^{n+1}}{(1-\theta_0)^{n+1} - (1-\theta_1)^{n+1}} = \frac{\theta_0 C_0 P(H_1)}{(\theta_1 - \theta_0)(1-C_0)P(H_0)} \tag{16}$$

As in Case 1, a uniform probability distribution is assumed both under the null hypothesis and the alternative hypothesis. In addition, an upper bound on the failure probability (θ_1) is defined under the alternative hypothesis. The required number of tests for different choices of $P(H_0)$, θ_0 and θ_1 are given in Table 3. The predefined confidence level C_0 is assumed to be constant equal to 0.99.

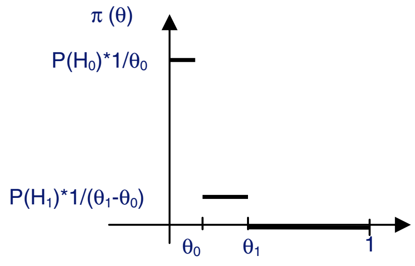


Fig. 2. The prior probability distribution used in Case 2

Table 3. Required number of tests for different choices of $P(H_0)$, θ_0 and θ_1, when a uniform prior probability distribution is assumed both under $P(H_0)$ and $P(H_1)$

θ_0	θ_1	**No. of tests $P(H_0)=0.01$**	**No. of tests $P(H_0)=0.6$**
0.01	0.05	776	284
0.001	0.05	5300	852
0.0001	0.05	30271	1242
0.000001	0.05	179002	1319
0.01	0.1	695	210
0.001	0.1	4602	510
0.0001	0.1	23804	639
0.000001	0.1	93500	659
0.01	0.25	598	131
0.001	0.25	3696	235
0.0001	0.25	15936	260
0.000001	0.25	38455	263
0.01	0.5	527	84
0.001	0.5	3025	124
0.0001	0.5	10853	131
0.000001	0.5	19412	131
0.01	1	458	50
0.001	1	2378	63
0.0001	1	6831	65
0.000001	1	9753	65

From Table 2 it can be seen that the number of tests, required to obtain adequate confidence in a software component, increases significantly when the upper bound for the failure probability decreases. Although the effect depends on the actual upper bound defined, some realistic test cases, where the upper bound was set to 0.1 and 0.05, indicate an increase in the number of required tests with a factor of respectively 10 and 20. These results can be explained by the Bayesian hypothesis testing approach, which uses the Bayes factor to find the number of tests required to satisfy the confidence level C_0. From equation (15), it can easily be seen that when θ_1 decreases, the Bayes factor also decreases. This means that the evidence in favour of the null hypothesis decreases, and that more tests must be conducted to satisfy the confidence level C_0.

Case 3:
A possible way to mitigate the effect of discontinuity in the prior probability distribution in Case 1 is to use a continuous probability distribution for θ over the entire interval (0, 1). To reflect prior beliefs, the beta distribution is often chosen (see Figure 3), since this distribution is a rich and tractable family that forms a conjugate family to the binominal distribution. A prior probability distribution for θ based on the beta distribution is shown in equation (17).

$$\pi(\theta) = \begin{cases} P(H_0)\dfrac{1}{P(H_0)}\dfrac{\tau(\alpha+\beta)}{\tau(\alpha)\tau(\beta)}\theta^{\alpha\text{-}1}(1\text{-}\theta)^{\beta.\text{-}1}, & \theta \le \theta_0 \\ P(H_1)\dfrac{1}{P(H_1)}\dfrac{\tau(\alpha+\beta)}{\tau(\alpha)\tau(\beta)}\theta^{\alpha\text{-}1}(1\text{-}\theta)^{\beta.\text{-}1}, & \theta > \theta_0 \end{cases} \tag{17}$$

As in the other two cases, it is assumed that no failures have been encountered during testing. Based on the binominal distribution in equation (5) and the prior belief about θ in equation (17), the Bayes factor, as defined in equation (7), can be calculated as shown in equation (18).

$$B = \frac{P(H_1)\displaystyle\int_0^{\theta_0}\theta^{\alpha\text{-}1}(1\text{-}\theta)^{\beta+n\text{-}1}d\theta}{P(H_0)\displaystyle\int_{\theta_0}^{1}\theta^{\alpha\text{-}1}(1\text{-}\theta)^{\beta+n\text{-}1}d\theta} \tag{18}$$

By combining equation (8) and (18), the number of tests required to satisfy the given predefined confidence level C_0, can be found. This is expressed in equation (19).

$$\frac{\displaystyle\int_0^{\theta_0}\theta^{\alpha\text{-}1}(1\text{-}\theta)^{\beta+n\text{-}1}d\theta}{\displaystyle\int_{\theta_0}^{1}\theta^{\alpha\text{-}1}(1\text{-}\theta)^{\beta+n\text{-}1}d\theta} = \frac{C_0}{1-C_0} \tag{19}$$

In situations were the prior belief regarding θ corresponds to the view that no failures have been detected during a hypothetical test, it is common to set the α-value in the prior beta distribution equal to 1. This is because the α-value in the beta distribution can be interpreted as the number of failures detected during testing, while the β-value can be interpreted as the total number of fault free tests performed during testing [1].

Table 4a shows the number of tests required to obtain adequate confidence in a software component, when α is constant equal to 1 and $P(H_0)$ varies between 0 and

0.99. The given predefined confidence level C_0 and upper bound θ_0 are assumed to be constant equal to 0.99 and 10^{-4}, respectively.

From Table 4a it can be seen that the number of required tests decreases towards 0 when the β-value increases.

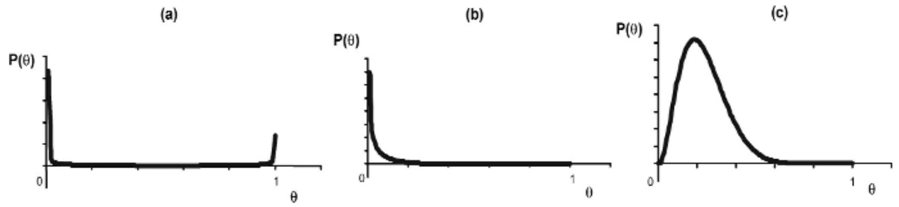


Fig. 3. Beta distribution with (a) α and β <1, (b) α<1 and β>1 and (c) α and β>1

This result can be explained by equation (19), were it easily can be seen that if the β-value increases, the number of remaining tests, *n*, required to satisfy the confidence level C_0 decreases accordingly. The first scenario in Table 4a, where $\beta \to 0$, represents classical statistical testing (black-box testing), while the second scenario represents using a uniform prior probability distribution for θ over the entire interval [0, 1]. In the last scenario, the prior belief regarding θ is equal to the posterior belief, and zero tests are needed to reach the required confidence level C_0.

Table 4. Required number of tests for $\theta_0 = 10^{-4}$ and $C_0 = 0.99$ when a continuous beta distribution are used a) with α = 1 and $P(H_0)$ varying from 0 to 0.99 and b) with $P(H_0) = 0.8$ and α varying from 10^{-4} to 10

$P(H_0)$	α	β	No. of tests
0.0	1	$\to 0$	46050
10^{-4}	1	1	46049
0.10	1	1054	44996
0.20	1	2231	43819
0.30	1	3567	42483
0.40	1	5108	40942
0.50	1	6931	39119
0.60	1	9163	36887
0.70	1	12039	34011
0.80	1	16094	29956
0.90	1	23025	23025
0.95	1	29956	16094
0.98	1	39119	6931
0.99	1	46050	0

(a)

$P(H_0)$	α	β	No. of tests
0.80	10^{-4} 0.0	$4*10^{-4}$	16536
0.80	1	0.071	16805
0.80	0.1	694	18993
0.80	0.8	13073	28985
0.80	1	16094	29956
0.80	3	42787	44477
0.80	10	125177	62645

(b)

Table 4b shows the required number of tests when $P(H_0)$ is constant equal to 0.8 and α varies between 10^{-4} and 10. As in Table 4a, the given predefined confidence level C_0 and upper bound θ_0 are assumed to be constant equal to 0.99 and 10^{-4}, respectively. From Table 4b it can be seen that the number of tests, required to satisfy the confidence level C_0, increases significantly when the α-value increases.

In addition, it can be seen that the total number of tests required by using the Bayesian hypothesis testing approach both can result in fewer as well as even more tests compared to classical statistical testing. The total number of required tests is highly dependent on the choice of α-value in the beta distribution.

4 Discussion

The results from the experiments in Chapter 3 show that the number of tests required to obtain adequate reliability in a software component is highly dependent on the choice of prior distribution in the Bayesian hypothesis testing approach. This prior distribution is an expression of prior belief in the reliability of a software component, and should be based on all existing information about the software component prior to testing. This chapter discusses how different evidences may influence this prior belief, and how this can be reflected in the prior distribution.

Some typical evidences that influence the reliability of a software component are: the quality of the producer, the development process, programming language, the complexity of the software, the type of software (in-house, COTS etc.), operating experience, etc. The sum of these evidences may not only have impact on how much one believe in a software component, but the different types of evidences may also influence the shape of the prior distribution between zero and one. In the following this is discussed, with reference to some software examples.

One aspect is that a software component can be correct, i.e. that it, in distinction from hardware components, can have a zero failure probability. This can be expressed by a θ_0 equal to zero, and a $P(H_0)$ that expresses the belief one has in this upper bound. If for instance the program is developed using clean room programming, or formal development methods, $P(H_0)$ may be high (close to one). However, there might be a suspicion that even if the program is correct according to specification, there is a possibility that the specification is incorrect. This suspicion should be expressed in the prior distribution under H_1. If there really is a specification error, this could lead to a failure at any moment of execution. It is therefore not unrealistic to assume that the failure probabilities under H_1 are evenly distributed. Alternatively, one could assume that if there is a specification error, this would lead to an immediate failure after execution start. In this case, the failure probabilities under H_1 should have a distribution that is large for θ near one, resulting in a kind of "bath-tub" curve for the distribution of θ (see Figure 3a).

A similar argument could be used for reuse of software. If a software component has been frequently used in one environment, with no failure, one can assume a large value of $P(H_0)$ for a low value of θ_0. However, if the same software component is used in another environment, there is a certain possibility for a systematic failure. One possible way to reflect this is by assuming a smaller value for $P(H_0)$ and a larger and more evenly distribution for the failure probabilities under H_1.

A different case would be for an in-house component, tailor made for its purpose, where the development was made in an unsystematic way. It is assumed that the component has been tested and debugged during development, and finally passed an acceptance test. In this case it is reason to believe that no obvious failures occur, so that the distribution for θ will be close to zero. On the other hand, in particular if the component is large and complex, there is a significant probability that there are one or more bugs hidden in the code that can cause failures in more rare situations. This should be reflected in the distribution for θ. A reasonable choice could be a uniform distribution for θ below a fairly low θ_0 and a fast decreasing distribution for θ above this value. The choice of θ_0, $P(H_0)$ and $P(H_1)$ should in this case depend on the belief one has in the development process and the thoroughness of acceptance test.

The intention of this chapter has been to stress the importance of selecting a prior distribution for θ on the basis of all available information, and give some ideas on how this can be done for different software components. Having chosen a reasonable shape of the distribution for θ, one could find a beta distribution that reflects this shape as well as possible. The use of beta distributions is, however, mainly chosen because of its computational convenience. Other more complex distributions may also be used.

5 Conclusion and Further Work

In this paper the use of Bayesian hypothesis testing, as one possible approach to find upper bounds for software failure probabilities, has been studied. Three different choices of prior probability distributions for a software component's failure probability have been evaluated, and their influence on the number of tests required to obtain adequate reliability in a software component has been investigated. The results from the experiments show that the number of required tests is highly dependent on the choice of prior distribution in the Bayesian hypothesis testing approach. In addition, it is shown that the total number of tests required by using this approach both can result in fewer as well as even more tests compared to classical statistical testing. It should be emphasized that it is not the Bayesian hypothesis testing approach that results in fewer required tests, but the underlying prior distribution for the software component's failure probability and the assumptions that are made. To choose a prior probability distribution for a software component's failure probability that correctly reflects one's prior belief is therefore of great importance.

Further work includes extending this approach to assess systems consisting of multiple software components. Of particular interest is the challenge of finding upper bounds for simultaneous failure probabilities, and thus making it possible to include dependency aspects in software reliability models.

Another important aspect, which needs to be studied further, is the difficulty of establishing descriptions of prior beliefs regarding single failure probabilities, as well as for simultaneous failure probabilities, where all available information prior to testing is taken into account. One approach to find these prior probability distributions that will be investigated is the application of Bayesian Belief Nets (BBN's).

Acknowledgement

This study is part of my PhD work at Østfold University College and the University of Oslo. I want to acknowledge: Rune Winther from Østfold University College, Gustav Dahll and Bjørn Axel Gran from Institute for Energy Technology for valuable comments and ideas.

References

1. Berger J. O.: *Statistical Decision Theory and Bayesian Analysis*. Springer verlag, 2nd edition, ISBN 0-387-96098-8, pp 118-166, 1980.
2. Cukic B., Gunel E., Singh H., Guo L.: The Theory of Software Reliability Corroboration. *IEICE Trans. on Information and Systems*, Vol. E86-D, No. 10, pp 2121-2129, Oct. 2003.
3. Eckhardt D. E., Lee L. D.: A theoretical basis for the analysis of redundant software subject to coincident errors. *NASA tech. Memo*, 86369, Jan. 1985.
4. Fenton N., Krause P., Neil M.: Software Measurement: Uncertainty and Causal Modeling. *IEEE software*, Vol. 19, No. 4, pp 116-122, July/Aug. 2002.
5. Frankl P., Hamlet D., Littlewood B., Strigini L.: Choosing Testing Method to Deliver Reliability. *Proc. of the 19th International Conference on Software engineering*, pp 68-78, May 1997.
6. Gran B. A.: *The use of Bayesian Belief Networks for combining disparate sources of information in the safety assessment of software based systems*. Thesis 2002:35, NTNU, Trondheim, Norway, 2002.
7. Hamlet D., Mason D., Woit D.: Theory of Software Reliability Based on Components, *International Conference on Software Engineering*, Vol. 23, pp. 361-370, 2001.
8. IEC 60880-2: *Software for Computers Important to Safety for Nuclear Power Plants – Part 2: Software aspects of defense against common cause failures, use of software tools and of pre-developed software*, December 2000.
9. Krishnamurthy S., Mathur A.: On the Estimation of Reliability of a Software System Using Reliabilities of its Components. Proc. *of the 8th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 1997.
10. Kuball S., May J., Hughes G.: Building a system failure rate estimator by identifying component failure rates. *Proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, pp 32-41, Nov. 1999.
11. Littlewood B., Miller D. R.: Conceptual Modeling of Coincident Failures in Multiversion Software. IEEE *Trans. on Software Engineering*, Vol. 15(12), pp 1596-1614, Dec.1989.
12. Littlewood B., Popov P., Strigini L.: Modelling software design diversity: a review. *ACM Computing Surveys*, Vol. 33, No. 2, pp 177-208, June 2001.
13. Lyu M. R. (editor): *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, ISBN 0-07-039400-8, 1995.
14. Miller K., Morell L. J., Noonan R. E., Park S. K., Nicol D. M., Murrill B. W., Voas J. W.: Estimating the probability of failure when testing reveals no failures. *IEEE Trans. on Software Engineering*, Vol. 18, No. 1, pp 33-43, Jan. 1992.
15. Poore J. H., Mills H. D., Mutchler D.: Planning and Certifying Software System Reliability. *IEEE software*, Jan. 1993.
16. Popov P., Strigini L., May J., Kuball S.: Estimating Bounds on the Reliability of Diverse Systems. *IEEE trans. on Software Engineering*, April 2003

17. Scott J. A., Lawrence J. D.: Testing Existing Software for Safety-Related Applications. *Lawrence Livermore National Laboratory*, prepared for U.S Nuclear Regulatory Commission, 1995.
18. Singh H., Cortellessa V., Cukic B., Gunel E., Bharadwaj V.: A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems. *Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE)*, 2001.
19. Smidts C., Cukic B., Gunel E., Li M., Singh H.: Software Reliability Corroboration. *Proc. of the 27'th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, 2002.

Justification of Smart Sensors for Nuclear Applications

Peter Bishop[1,2], Robin Bloomfield[1,2], Sofia Guerra[2], and Kostas Tourlas[2]

[1] CSR, City University
[2] Adelard, Drysdale Building, 10 Northampton Square, London, EV1V 0HB, UK
{pgb, reb, aslg, kt}@adelard.com

Abstract. This paper describes the results of a research study sponsored by the UK nuclear industry into methods of justifying smart sensors. Smart sensors are increasingly being used in the nuclear industry; they have potential benefits such as greater accuracy and better noise filtering, and in many cases their analogue counterparts are no longer manufactured. However, smart sensors (as it is the case for most COTS) are sold as black boxes despite the fact that their safety justification might require knowledge of their internal structure and development process. The study covered both management aspects of interacting with manufacturers to obtain the information needed, and the technical aspects of designing an appropriate safety justification approach and assessing feasibility of a range of technical analyses. The analyses performed include the methods we presented at Safecomp 2002 and 2003.

1 Introduction

Sensors for nuclear applications have been relatively simple analogue devices with known performance properties and known failure characteristics. However, the sensor industry is increasingly using microprocessor-based "smart sensors". Smart sensors can achieve greater accuracy, better noise filtering together with in-built linearisation, and provide better on-line calibration and diagnostics features. So, given the difficulty in obtaining replacement analogue sensors, and the potential benefits of smart sensors, it is important that the nuclear industry develops a suitable approach for justifying the use of smart sensors in systems important to safety (SIS).

Smart sensors are a specific form of COTS (commercial off-the-shelf) product. COTS products are normally sold as a "black box" where there is no knowledge of the internal structure. However, their safety justification might require knowledge of the internal structure and development process. The justification of sensors is an increasing problem because the software constitutes a valuable intellectual investment, and the civil nuclear companies purchase sensors in small quantities.

This paper presents the results of a research study sponsored by the UK nuclear industry into methods for justifying smart sensors. The project has covered both management and technical issues.

- From a management perspective, we examined the issues involved in interacting with the suppliers to gain the information needed for the justification. We also addressed the need for a sustainable long-term approach for the justification of smart sensors that is acceptable to both suppliers and customers.

- From a technical perspective, we need an assessment approach that is both proportionate and feasible to apply and is commensurate with the SIL of the intended application(s). The approach should also be related to existing assurance requirements for computer-based systems in nuclear application (e.g. HSE SAPs [1] and the British Energy PES Guidelines [2]) and should address the concerns related to the black box assessment of smart sensors (and other COTS products).

The paper is structured as follows. In Section 2 we describe the interactions with the manufacturers that took place in the scope of this project, as well as the main issues identified during these interactions. Section 3 summarises three different viewpoints of the safety justification of smart sensors, including a goal-based approach that is expanded in Section 4 and vulnerability assessment summarised in Section 5. Section 6 describes the analyses performed on one of the smart devices obtained, while Section 7 relates these analyses with the three approaches from Section 3. The conclusions are presented in Section 8.

2 Relationships with Smart Sensor Manufacturers

2.1 Obtaining Software, Supporting Data and Company Culture

Obtaining the software was a lengthy process spread over many months, involving several contacts, phone calls and negotiation of conditions for non-disclosure of the results. However, once the relationship was established, there were fewer difficulties in providing further data on the devices or even the software of other devices.

As a result of these negotiations, we successfully obtained smart sensor software from two manufacturers, and we were offered the possibility of obtaining another example by a third manufacturer. In addition, the manufacturers supplied, to varying degrees, design documentation and additional data such as process and reliability data and certificates that could be used to support the safety justification of the devices.

It must be borne in mind, however, that the software was provided on the understanding that it was a part of a research project undertaken by a specialist third party. Whether a similar level of access would be provided to end-users on a routine basis remains an open question.

The nature of the relationships with the smart sensor manufacturers was quite distinct, probably as a result of the key markets they target and the management structure and particular characteristics of the companies. It is clear that smart sensor suppliers can have different company cultures, and this can have a major impact on the feasibility of gaining access to the source code and subsequently a successful assessment. Some pertinent questions to assess the likely success of the interaction might be:

1. How does the organisation deal with this type of interaction? Are there precedents in the past?
2. Who is the gatekeeper and what is their role in the organisation? (a "gatekeeper" is an official point of contact in the company who decides whether to allow requests to pass to higher management):

- Do they have access to design authority?
- What are the lines of communication?
- How frequently and with whom are meetings available to clarify issues?

3. How difficult would access to management be?
4. To what extent does the organisation see the relationship as benefiting them?
   - What are the main market sectors?
   - Is nuclear industry a main market, a niche or a distraction?
   - Are the benefits of engagement seen only in terms of future sales?
   - Does the organisation see benefits of interaction from a technical or process improvement point of view?
5. What is the attitude to confidentiality?
   - Is a standard non-disclosure agreement sufficient?
   - Is the confidentiality agreement with the assessor company or with named people from the company?

2.2 Long Term Issues

Despite the differences between the supplier approaches, some common long-term issues emerged:

1. The suppliers expressed concerns about the effort and cost needed for routine justifications of smart sensors—the nuclear industry is a small market compared to other sectors, and the effort might be excessive relative to the potential sales.
2. Both suppliers are generally in favour of an "assurance package" of additional information that is paid for by the customer.

The big issue is what this package should contain and whether it will be acceptable to a wide range of customers (e.g. the nuclear industry or beyond). There is a UK initiative to define a framework of IEC 61508-conformant documentation about the development processes for a device; this has the potential to form part of an assurance package. The SIREP sensor assessment could also be used in support of the functional and hardware performance of the sensor. Further work would be needed to describe the full content of such a package. The main area of weakness is the "black box" nature of such evidence, and greater confidence could be obtained if there was some knowledge of the internal design and implementation of the device (e.g. "grey box" information like design documents, or white box information like source code).

Independent certification is used to provide "black box" evidence for performance (e.g. accuracy) and environmental withstand (e.g. electrical isolation, temperature). Evidence also exists for hardware reliability and safe failure fraction (e.g. using hardware reliability models and FMEDA). But independent assessment of software (like a TUV assessment for compliance to IEC 61508) is not routinely performed. This is thought to be quite costly especially if it has to be updated with each revision.

It is clear there is a perceived tension between the need for suppliers to maintain confidentiality and the user's requirement to provide sufficient evidence to demonstrate safety. In principle the licensee should always be able to check the evidence and the details of the analysis performed. This might not be feasible for a compliance certification approach where a certificate is provided but the rationale and detailed evidence is not. By contrast "black box" system certification against test standards seems less of a problem as the specific tests undertaken are clearly defined. Perhaps third party software evidence would be more acceptable if a common set of software analyses and tests were defined that could be subjected to external audit.

3 Safety Justification Approaches

There are different strategies that can be deployed in the safety justification of smart sensors. The three main approaches can be characterised as a "triangle" of:

- Justification via a set of claims about the system's safety behaviour.
- The use of accepted standards and guidelines.
- An investigation of known potential vulnerabilities of the system.

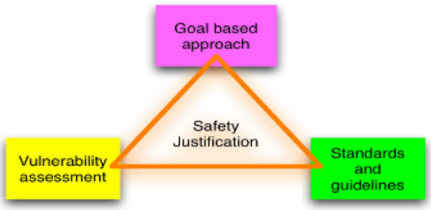


Fig. 1. Safety case approaches

The first approach is goal-based—where specific safety goals for the systems are supported by arguments and evidence at progressively more detailed levels. The second approach is based on demonstrating compliance to a known safety standard. The final approach is a vulnerability-based argument where it is demonstrated that potential vulnerabilities within a system do not constitute a problem—this is essentially a "bottom-up" approach as opposed to the "top-down" approach used in goal-based methods. These approaches are not mutually exclusive, and a combination can be used to support a safety justification, especially where the system consists of both off-the-shelf (OTS) components and application-specific elements.

In the past, safety justifications tended to be implicit and standards-based—compliance to accepted practice was deemed to imply adequate safety. This approach works well in stable environments where best practice was supported by extensive experience, but with fast moving technologies, a more explicit goal-based approach has been advocated, which can accommodate change and alternative strategies to achieve the same goal.

4 Goal-Based Safety Justification

4.1 Goal-Based Justification of COTS Products

Goal-based approaches are often used in safety justifications ([3], [4], [5], [6]). This is a flexible approach as it focuses directly on the safety requirements for the sensor and can be related to a range of different safety standards by identifying how the standards' requirements support the various claims.

HSE guidance on COTS/SOUP ([7], [8]) recommends that the goals for a computer-based system are related to factors that directly affect safety, e.g.:

- functional behaviour
- accuracy
- reliability and availability
- fail-safe behaviour
- time response

A similar set of attributes could be identified for a smart sensor component of a system. Moreover, we have to recognise that a smart sensor justification is part of a larger safety justification for a "System Important to Safety" (SIS).

1. The justification of the component can be used to show that the component "does what it says on the tin". This is application independent.
2. The SIS safety justification has to show that the component is suitable for the application context and satisfies any constraints.

For example, there could be a component claim that a smart sensor is accurate to 10^{-3} and an application-level claim of 10^{-2} for the accuracy of some computation involving the combination of several measured values. Alternatively, the sensor might explicitly include functionality that is not needed (e.g. support for different types of resistance thermometer). In this case we need to show in the SIS safety justification that the unwanted functions are not activated. In addition there may well be functionality that is not declared as part of the product.

The extent of evidence to justify specific sensor properties will vary with the required integrity level, and the type of evidence required may also need to change with the integrity. At lower integrity levels there is greater emphasis on "black box" evidence (externally observed behaviour) and evidence of development process. Yet, it is necessary to look "inside the box" for greater assurance and higher integrity applications. Indeed the public consultation on the HSE study showed a consensus that the assessment of critical SOUP should include white box assessment.

4.2 Key Goals

In a goal based approach we identify the key attributes required for a smart sensor, and then seek arguments and evidence to show these goals are met. Clearly many properties like environmental limits (temperature, humidity, supply voltage) and resistance to interference (RFI, EMI, mains noise) are only dependent on the hardware and can be justified in the same way as conventional hardware. However, other properties will depend on a combination of hardware and software and may therefore require different or additional supporting evidence to address software concerns.

We focused on the properties of the system that involve software. Based on the specifications of the two suppliers and more general consideration of smart sensor requirements, the following set of goals was identified that are likely to involve software in the smart sensor, where the values used correspond to those of the temperature transmitter specification. This could be extended to include functional capabilities of the device.

Table 1. Smart sensors properties for goal-based approach

Ref	Sensor property
1	Output conversion accuracy better than 0.03% under stated conditions
2	Sample rate < 128 ms, 10%-90% step response < 256 ms
3	Safe failure fraction >0.72
4	MTTF > 2.5 years (average)
5	Configuration and calibration errors are minimised
6	Smart sensor behaviour is predictable
7	Smart sensor properties 1-6 can be maintained for next 10 years

Note these goals are application independent, i.e. defined by the supplier. The numerical values will vary with the actual sensor, but these properties are directly relevant to the safety justification of the SIS as a whole. The system implementor has to define higher-level system goals (e.g. timeliness and accuracy of the overall system) and demonstrate that the properties of the sensor support the SIS safety justification.

5 Vulnerability Assessment

The vulnerability assessment focused on a number of concerns with a purely black-box based approach. Discussion of these concerns is not included in the paper, but included adequacy of functional testing, software security vulnerabilities and malicious code among others. In subsequent analysis we expanded the final category of "non-predictability" to consider specific sources of non-predictability, namely:

- concurrent interaction problems (non-atomic updates, deadlocks, etc.)
- non-initialisation of data
- data overflow
- variable time response (data dependent timing, infinite loops, etc.)

The black box concerns identified above could result in unexpected behaviour even if the equipment appears to conform to specification when functional testing of the "black box" is performed. Such concerns can be used as a checklist to determine whether the risks of potential vulnerabilities have been addressed (e.g. using additional grey-box or white-box evidence). The fundamental limitation of black-box analysis is the extent to which the testing/field experience profile used mirrors the actual use in the new application.

6 Analysis of a Smart Device

In Section 3 we described different aspects of a safety justification of a smart sensor, but no supporting evidence was presented. In this section we describe the technical analyses performed in the project concerning a specific sensor product: a temperature transmitter. The aim of this work was to assess the feasibility of a variety of techniques and analyses to support the safety justification of the temperature sensor.

The microprocessor is programmed in assembly language. The software comprises

- 6000 lines of assembly code (including comments)
- 7 kilobytes of binary code

The source code has been subjected to a range of assessments to support its safety justification, namely:

- Code criticality analysis. Identification of the code essential to operation and obsolescent code.
- Code structure analysis. Identification of concurrent program threads and checks for safe exchange of data between threads and absence of deadlocks.
- Code integrity assessment. Check for defective code constructs (e.g. array bound overflow, divided by zero, dead code, stack overflow).
- Redundant code analysis. To identify if there is any unexpected code.
- Failure integrity analysis. Assessment of the failure integrity features to check whether failures result in a safe state.
- Predictable execution assessment. Assessment of whether the ordering of software functions is well defined, of the input-output conversion accuracy and whether the execution time has a predictable upper execution time bound.
- The development of a strategy for statistical testing (not described in this paper).

The feasibility of performing such analyses depends on the code size, structure, and programming language. There is more tool support for high level languages (like C) for performing analyses. However, we have made use of assembler level software simulators to perform direct testing of internal functions within the software and to evaluate the code coverage and code execution times.

6.1 Code Criticality Analysis

We performed software criticality analysis [9] to assess the importance to safety of various components within the software. This showed that the code is separated into the main conversion function and a calibration function. The calibration code can only modify the calibration parameters used by the main code, and hence could be viewed as less critical as it is not executed in normal operation. Most of the analysis effort (such as the timing analysis) was focused on the main conversion function.

6.2 Structural Analysis and Concurrency Analysis

In order to perform evaluations of the sensor software it was first necessary to understand the overall software architecture. To gain an understanding of the structure, we reviewed both the software documentation and the source code.

As part of the structural analysis, we identified concurrent program threads and checked for safe exchange of data between threads and absence of deadlocks.

From the architecture analysis, we identified the variables that passed information between the threads. It is important that these updates are "atomic", i.e. cannot be seen by another thread in a partially updated state. For example, if a two byte variable

is updated from "00, FF" to "01, 00", it could transiently have the value "01, FF" or "00, 00" (depending on the update order of the bytes). Clearly it is undesirable that the variable can be seen in this transient state, and the software should ensure that all updates are atomic. The software was manually reviewed to check for this.
We also verified that these atomic updates do not result in deadlock. Whenever interrupts are set, they are always released.

6.3 Code Integrity Assessment

Integrity static analysis [10] focused on structural faults in the software, in particular the internal integrity of the code and the intra-component integrity.

The extensive field experience of the device being analysed is likely to precipitate the detection of most large and obvious faults during typical execution of the program. However, specific vulnerabilities of the languages used have a less frequent manifestation and are more likely to remain undetected. Integrity static analysis focuses on what we called intrinsic faults, i.e. faults that may be recognised as such independently of any requirements specification. This includes use of out of bound array indexes, use of illegal pointers, use of non-initialised variables, violations of assertions, permanent loss of resources, insufficient resources, dead locks, non-deterministic behaviour and non-controlled access to shared variables.

Our approach to the analysis of the assembler code can be grouped into two main categories:

- Direct analysis of the assembler code, including analysis of the control flow supported by model checking, dead code checks and semantic analysis.
- Translation of the assembler code into C (by a Perl script combined with manual translation) and analysis of the translated code using CodeSurfer and Safer C.

6.4 Redundant Software Analysis

The software was reviewed for redundant functions. The only redundant function found was a display function that apparently drives a local LCD display on the sensor—however the sensor supplied to us has no LCD display. We suspect that this is because the same code is used to drive a family of sensors including some with built-in displays. From a configuration management and support perspective, it is desirable to use common code. There is no real objection to this apart from the fact that it must be demonstrated that the software does not "hang" waiting for a response from the display hardware and it conflicts with IEC 60880, a fundamental standard for reactor protection systems. We do not think this is a problem as the unit supplied to us functions correctly without the LCD display yet a more formal safety justification would need to document this and to justify the non-compliance with IEC 60880.

6.5 Failure Integrity Analysis

It is important that failures of the hardware and the software result in a safe state. An analysis was performed to identify what fault detection and fault handling features were present in the software. Analysis showed that internal software failures and delays were trapped by a hardware watchdog. An analysis of the software also showed that it contained checks on the integrity of the hardware (as required in IEC 61508 Part 2 Annex A1). These included software checks for breaks in probe wiring, memory integrity, and the integrity of the analogue to digital interface. Failure integrity was also enhanced by using a software-generated oscillating binary output to drive the analogue output circuit. If failures occurred in the processor, output interface, or software, the output bit would become static and result in an out-of-range output signal.

6.6 Functional Analysis-Accuracy Analysis

The functional software analyses focused on checking whether the software maintains sufficient accuracy. The functional analyses were performed using the Cosmic suite of tools for assembling, simulated execution, debugging and profiling of the code for a micro-controller.

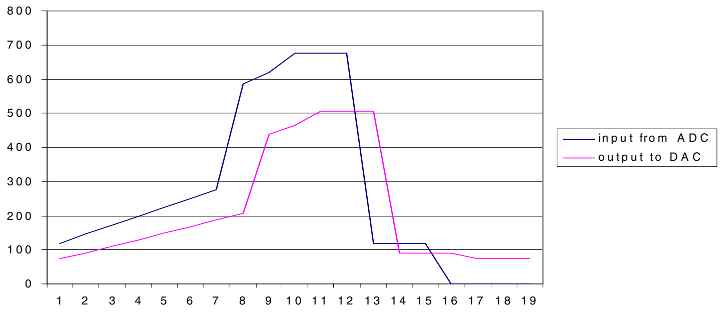


Fig. 2. Input vs. simulated output (mV operation, no linearisation)

The smart sensor software can be configured to operate a number of different modes, or types of measurement. We configured the software to perform a simple linear scaling of the input, and then simulated the execution of the smart sensor code on a PC using the COSMIC simulator tool.

Figure 2 plots the simulated input (X) alongside the converted output (Y). (The units along the horizontal axis count the samples.) The plot confirms a precise match between expected and actual behaviour: the output closely follows the overall envelope of the input, albeit with its slope adjusted by a factor of 0.75, until the input is plunged to zero. The final drop in the input simulates an error condition such as a broken wire or other malfunction of the analogue input. When such errors occur, the output should be forced to the minimum output signal value of 75—and this behaviour is observed in the simulation. Figure 3 plots the magnitude of the rounding error for the first 15 samples in the input sequence. The rounding error is defined as the difference between the value produced by the conversion equation and the "ideal" value.

The plot shows the rounding error to be bounded within +/-0.5 of one output unit. This represents an error in the order of 0.0125% compared to the output of an "ideal" analogue converter. This is in line with our expectations based on inspection of the code analysis, which showed that high precision arithmetic routines are used in the conversion calculation.

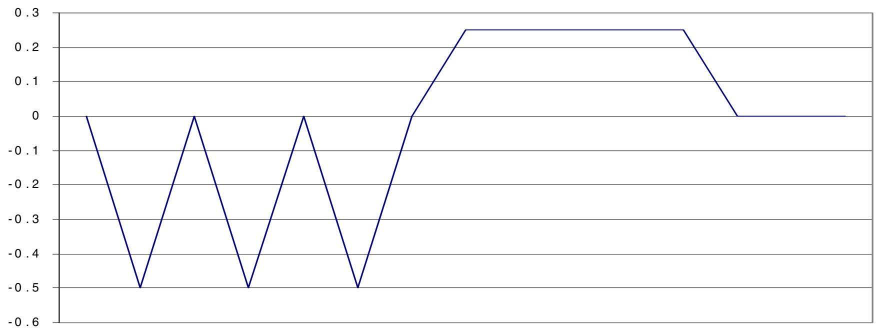


Fig. 3. Magnitude of rounding error

6.7 Timing Analysis

It is important that the sensor has predictable timing, so the measurements are updated at the specified intervals. Inspection of the code showed that the input-output conversion was performed in a simple cyclic loop. This helped to ensure that the sampling of the input and updating of the output occur at predictable rates.

The overall strategy followed by the analysis was to:

- Obtain simulated execution timings (using the COSMIC tool) for the software conversion.
- Derive worst-case estimates for the amount of time the micro-controller spends communicating instructions and data with the ADC. These are derived based on timing data available on the serial interface between the micro-controller and the ADC, as well as the ADC's internal operation.
- Combine the two, bearing in mind potential overlaps of activity, and check the combined timings against the claimed targets.

The complete timing estimates were derived from the information above to substantiate the original claims, and it was concluded that the software performs within its timing specifications, even under conservative and pessimistic assumptions.

7 How Evidence Fits the Safety Justification Approach

This section relates the analysis evidence obtained from the sensor (Section 6) with the goal-based approach (Section 4) and black-box concerns (Section 5). Note that the overall "triangle" of safety justification also included an assessment of compliance to the British Energy PES guidelines [2]. However, standards compliance will not be discussed further in this paper.

7.1 Goal-Based Approach

A goal-based approach for a smart sensor would focus on demonstrating key properties of the sensor that are relevant to the safety justification of the system. In Section 4 we identified a set of sensor properties that represent the top-level goals of the smart sensor justification. This section discusses the evidence that can be used to show that the goals have been met.

- Evidence for goals 1 and 2 (accuracy and response time) can be obtained from black-box functional tests (e.g. SIREP or statistical tests). However, it would also be possible to strengthen the justification by using diverse evidence from white box analyses. For example, the analyses of software accuracy and timing undertaken in Sections 6.6 and 6.7 could be part of an alternative white-box argument that the overall combination of hardware and software will satisfy the top-level goals.
- Goals 3 and 4 (safe failure fraction and MTBF) can also be demonstrated by diverse means. With adequate reporting of operational defects by end users, it might be possible to analyse the field experience to directly measure the MTBF and safe failure fraction. However, it is recognised that there are many uncertainties in such estimates, and it is also possible to make an analysis of the hardware and diagnostic features to make alternative estimates. In our particular example, the supplier provided a FMEDA assessment to support the claims.
- Goal 5 (configuration and calibration integrity) is important as errors in configuration and maintenance are a significant source of safety problems. Clearly the configuration interface should seek to minimise errors. We have analysed the design features that protect against random corruption using evidence from the integrity analysis, but this does not cover aspects such as the usability of the interface. This might be addressed by a human factors assessment.
- Goal 6 (predictable behaviour) is generally important within the safety justification. If the behaviour is predictable, the results of black-box tests can be used with more confidence because the behaviour is likely to be repeatable. To some extent predictability can be justified by field experience, but most of the evidence is likely to be based on analysis of the architecture and source code. The structural analyses described in Section 6.2 give confidence that some known architectural problems (like deadlocks) do not exist, and the vulnerability assessment summaised in Section 7.2 also help to demonstrate predictability. In addition, the integrity analyses we performed showed that certain types of source code faults were absent. These are however quite detailed analyses that may be impractical for lower integrity sensors, so an alternative source of evidence might be grey-box knowledge of the design process. For example, the process might include rules about the design of the system for deadlock avoidance and predictable timing or procedures to perform static analysis to detect various types of software fault.
- Goal 7 (long-term dependability) is also important for the long-term safety of the overall system. Smart sensor products are subject to change (e.g. of hardware components and software functionality) and might also cease to be manufactured or supported. We need evidence of availability and support, but we also require assurance that changes do not affect any of the other safety goals. This is partly addressed by the technical analysis given in Section 6, but knowledge of the supplier's process can give some confidence that the changes will do not affect the behaviour of the sensor.

It can be seen that a goal-based approach helps to identify what evidence helps to support the safety justification. We can also see that multiple forms of evidence could be deployed to support the same goal, and this gives flexibility in the justification approach. With higher integrity applications, we might expect greater use of diverse evidence and arguments. We also note that evidence about the process is only indirectly related to the functional properties of the sensor, but is likely to be highly relevant in ensuring that the observed behaviour is predictable and will be maintained in subsequent upgrades.

7.2 Vulnerability Assessment

This section summarises how specific concerns have been or could be addressed to support the justification of a smart device. These would support goal 6 in Section 7.1.

Table 2. Summary of vulnerability analysis

Concern	Activity/method	Observation
Adequacy of functional testing	Functional analysis	Partially tested through simulation to establish functional requirements and accuracy were met.
Housekeeping code	Code inspection & structural analysis	No problems identified.
Detection of infeasible paths	Dead-code analysis, simulation	Dead-code analysis identified few cases of inaccessible code. Simulation uncovered no infeasible paths.
Fault-detection code	Code inspection and structural analysis	Inspection revealed no problems.
Fault-tolerance	Code inspection and structural analysis	The design of the product emphasises fail-safety over fault-tolerance. The software has good fault integrity behaviour.
Time-based events	Code inspection, timing analysis	No calendar time-based events were found.
Counters	Structural analysis, code analysis	No cases of potential counter overrun were found in the code.
Malicious code	Code inspection	None found.
Unwanted requirements	Documentation review, code analysis	Some unused code was found, but it was used by other models in the same sensor family.
Security vulnerabilities	Code inspection	None found in the software running on the product itself.
Complexity	Structural analysis, code inspection	The software architecture is simple. Main sources of complexity are the relatively dense branching structure of the code and the bit-field encoding of configuration data.
Concurrent interactions	Structural analysis, concurrency analysis	The sequence of functions performed is predictable. Concurrency analysis shows that data exchange between threads is limited and the data exchanges are atomic.
Initialisation errors	Structural analysis	Not analysed in detail.
Data overflow	Structural analysis	Not analysed in detail. However, most of the data used are fixed multi-byte variables, so overflows are unlikely.
Variable time response	Structural analysis Timing analysis	Analyses have shown the code will execute within the specified response time.

8 Conclusions

In this section we present the conclusions of the research study into methods for justifying smart sensors that this paper is describing. While some of the conclusions follow directly from the work here described, others have arisen from work not specifically mentioned in the paper. The conclusions of the project are given below.

8.1 Relationship with Sensor Manufacturers

- End users should expect different degrees of co-operation, both within a single manufacturer's organisational hierarchy and across different manufacturers.
- It remains an open question whether end users will be offered as much access to software as we have in the course of this project. However, the level of co-operation might increase with the potential of a real order, which would not be the case in a research project.
- We need to be specific about what needs to be demonstrated in an assurance package that is provided by a supplier. Some form of assurance about the product is needed, principally to confirm predictability of behaviour and integrity in the presence of hardware failures. This evidence would typically be a combination of "grey box" (e.g. documentation on the software design) and white box evidence (based on analysis of the actual software), as greater confidence could be obtained if there was some knowledge of the internal design and implementation of the device.

8.2 Safety Justification Approaches

- Justification may follow a goal-based approach, aim to demonstrate compliance of the product with industry guidelines, or seek specifically to address areas of concern that a typical black box assessment would not cover. These three viewpoints should be considered simultaneously as they complement each other.
- Goal-based approaches seem particularly suitable to smart sensor products as they focus directly on the safety requirements, can be tailored to standards and offer flexibility and a certain potential for reuse of arguments and evidence.
- We have addressed specific concerns about the shortcomings of black-box assessment by assessing their applicability to smart sensor products similar to the ones we have examined. Grey-box and white-box evidence can compensate for the limitations of black-box assessment.

8.3 Smart Sensor Evaluation Methods

- We have undertaken a range of structural, accuracy and timing analysesbased on documentation and the source code, and these have proved to be "easy wins".
- Such analyses might be improved with appropriate tool support (e.g. code and data flow dependency analysis, worst case time analysis), although this might be problematic for assembler based systems.
- We have shown that it is feasible to simulate execution to test specific attributes such as accuracy, test coverage and timing. However it is likely that these tests would have been easier to implement during development where there is no need to simulate the action of attached peripherals.
- It was technically feasible to perform a range of evaluations on the smart sensor (even though it was written in assembler). This was largely due to the relatively small size of the software and simplicity of the design.

8.4 Further Work

There are many pragmatic issues that have not been addressed within the current study (e.g. what evidence is needed for a given integrity level and who should perform the evaluation). However we hope that the feasibility studies performed in this project can contribute to the development of a common approach to the justification of smart sensors in the nuclear industry.

Acknowledgements

This work was partly funded under the HSE Generic Nuclear Safety Research Programme under contract 40072849. The views expressed in this report are those of the authors and do not necessarily represent the views of the members of the CINIF or the Health and Safety Commission/Executive. The CINIF does not accept liability for any damage or loss incurred as a result of the information contained in this paper.

References

[1] Nuclear Safety Directorate, "Safety assessment principles for nuclear plants", http://www.hse.gov.uk/nsd/saps.htm

[2] L.A. Winsborrow, A.R. Lawrence, "Guidelines for Using Programmable Electronic Systems in Nuclear Safety and Nuclear Safety-Related Applications", British Energy 2002.

[3] P.G. Bishop and R.E. Bloomfield, "The SHIP Safety Case—A Combination of System and Software Methods", SRSS95, Proc. 14th IFAC Conf. on Safety and Reliability of Software-based Systems, Brugge, Belgium, 12-15 September 1995.

[4] P.G. Bishop and R.E Bloomfield, "A Methodology for Safety Case Development", Safety-critical Systems Symposium, Birmingham, UK, Feb 1998.

[5] CEMSIS project, http://www.cemsis.org

[6] J.A. McDermid, "Support for safety cases and safety argument using SAM", Reliability Engineering and Safety Systems, Vol. 43, No. 2, 111-127, 1994

[7] C.C.M. Jones, R.E. Bloomfield, P.K.D. Froome and P.G. Bishop, "Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP)", Report No: CRR337 HSE Books 2001 ISBN 0 7176 2011 5 http://www.hse.gov.uk/research/crr_pdf/2001/crr01337.pdf

[8] P.G. Bishop, R.E. Bloomfield and P.K.D. Froome, "Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications", Report No: CRR336 HSE Books 2001 ISBN 0 7176 2010 7 http://www.hse.gov.uk/research/crr_pdf/2001/crr01336.pdf

[9] P.G. Bishop, R.E. Bloomfield, T.P. Clement and A.S.L. Guerra, "Software Criticality Analysis of COTS/SOUP" Safecomp 2002, Catania, September 2002.

[10] P.G. Bishop, R.E. Bloomfield, T.P. Clement, A.S.L. Guerra, and C.C.M. Jones, "Integrity Static Analysis of COTS/SOUP" Safecomp 2003, Edinburgh, September 2003.

Evolutionary Safety Analysis: Motivations from the Air Traffic Management Domain

Massimo Felici

LFCS, School of Informatics, The University of Edinburgh, Edinburgh EH9 3JZ, UK
`mfelici@inf.ed.ac.uk`
`http://homepages.inf.ed.ac.uk/mfelici/`

Abstract. In order realistically and cost-effectively to realize the ATM (Air Traffic Management) 2000+ Strategy, systems from different suppliers will be interconnected to form a complete functional and operational environment, covering ground segments and aerospace. Industry will be involved as early as possible in the lifecycle of ATM projects. EUROCONTROL manages the processes that involve the definition and validation of new ATM solutions using Industry capabilities (e.g., SMEs). In practice, safety analyses adapt and reuse system design models (produced by third parties). Technical, organisational and cost-related reasons often determine this choice, although design models are unfit for safety analysis. Design models provide limited support to safety analysis, because they are tailored for system designers. The definition of an adequate model and of an underlying methodology for its construction will be highly beneficial for whom is performing safety analyses. Limited budgets and resources, often, constrain or inhibit the model definition phase as an integral part of safety analysis. This paper is concerned with problems in modeling ATM systems for safety analysis. The main objective is to highlight a model specifically targeted to support evolutionary safety analysis.

1 Introduction

The future development of Air Traffic Management (ATM), set by the ATM 2000+ Strategy [9], involves a structural revision of ATM processes, a new ATM concept and a systems approach for the ATM network. The overall objective [9] is, *for all phases of flight, to enable the safe, economic, expeditious and orderly flow of traffic through the provision of ATM services, which are adaptable and scalable to the requirements of all users and areas of European airspace.* This requires ATM services to go through significant structural, operational and cultural changes that will contribute towards the ATM 2000+ Strategy. Moreover, from a technology viewpoint, future ATM services will employ new systems forming the emergent ATM architecture underlying and supporting the European Commission's Single European Sky Initiative.

ATM services, it is foreseen, will need to accommodate an increasing traffic, as many as twice number of flights, by 2020. This challenging target will require the cost-effectively gaining of extra capacity together with the increase of safety levels [28,29]. Enhancing safety levels affects the ability to accommodate increased traffic demand as well as the operational efficiency of ensuring safe separation between aircrafts. Suitable safe conditions shall precede the achievement of increased capacity (in terms of accommodated flights). Therefore, it is necessary to foreseen and mitigate safety issues in aviation where ATM can potentiality deliver safety improvements. Introducing safety relevant systems in ATM contexts requires us to understand the risk involved in order to mitigate the impact of possible failures. Safety analysis involves the activities, i.e., definition and identification of system(s) under analysis, risk analysis in terms of tolerable severity and frequency, definition of mitigation actions, that allow the systematic identification of hazards, risk assessment and mitigation processes in critical systems [24,37].

Diverse domains (e.g., nuclear, chemical or transportation) adopt safety analyses that originate from a general approach [24,37]. Recent safety requirements, defined by EUROCONTROL (European organization for the safety of air navigation), imply the adoption of a similar safety analysis for the introduction of new systems and their related procedures in the ATM domain [8]. Unfortunately, ATM systems and procedures have distinct characteristics[1] (e.g., openness, volatility, etc.) that expose limitations of the approach. In particular, the complete identification of the system under analysis [22] is crucial for its influence on the cost and the effectiveness of the safety analysis. Some safety-critical domains (e.g., nuclear and chemical plants) allow the unproblematic application of conventional safety analysis. Physical design structures constrain system interactions and stress the separation of safety related components from other system parts. This ensures the independence of failures. By contrast, ATM systems operate in open and dynamic environments where it is difficult completely to identify system interactions. For instance, there exist *complex interactions*[2] between aircraft systems and ATM safety relevant systems [31]. Unfortunately, these complex interactions may give rise to catastrophic failures. The accident (1 July 2002) between a BOEING B757-200 and a Tupolev TU154M [5], that caused the fatal injuries of 71 persons, provides an instance of unforeseen complex interactions. These interactions triggered a catastrophic failure, although all aircraft systems were functioning properly [5]. Hence, safety analysis has to take into account these complex interaction mechanisms (e.g., failure dependence, reliance in ATM, etc.) in order to guarantee and even increase the overall ATM safety as envisaged by the ATM 2000+ Strategy.

This paper is concerned with limitations of safety analysis with respect to evolution. The paper is structured as follows. Section 2 describes safety analysis

[1] *"There are some unique structural conditions in this industry that promote safety, and despite complexity and coupling, technological fixes can work in some areas. Yet we continue to have accidents because aircraft and the airways still remain somewhat complex and tightly coupled, but also because those in charge continue to push the system to its limits. Fortunately, the technology and the skilled pilots and air traffic controllers remain a bit ahead of the pressures, and the result has been that safety has continued to increase, though not as markedly as in early decades."*, p. 123, [31].

[2] *"Complex interactions are those of unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible."*, p. 78, [31].

in the ATM domain. Unfortunately, ATM systems, procedures and interactions expose limitations of safety analysis. Section 3 proposes a framework that enhances evolutionary safety analysis. Section 4, finally, draws some conclusions.

2 Safety Analysis in ATM

ATM services across Europe are constantly changing in order to fulfil the requirements identified by the ATM 2000+ Strategy [9]. Currently, ATM services are going through a structural revision of processes, systems and underlying ATM concepts. This highlights a systems approach for the ATM network. The delivery and deployment of new systems will let a new ATM architecture to emerge. The EUROCONTROL OATA project [35] intends to deliver the Concepts of Operation, the Logical Architecture in the form of a description of the interoperable system modules, and the Architecture Evolution Plan. All this will form the basis for common European regulations as part of the Single European Sky.

The increasing integration, automation and complexity of the ATM System requires a systematic and structured approach to risk assessment and mitigation, including hazard identification, as well as the use of predictive and monitoring techniques to assist in these processes. Faults [23] in the design, operation or maintenance of the ATM System or errors in the ATM System could affect the safety margins (e.g., loss of separation) and result in, or contribute to, an increased hazard to aircrafts or a failure (e.g., a loss of separation and an accident in the worst case). Increasingly, the ATM System relies on the reliance (e.g., the ability to recover from failures and accommodate errors) and safety (e.g., the ability to guarantee failure independence) features placed upon all system parts. Moreover, the increased interaction of ATM across State boundaries requires that a consistent and more structured approach be taken to the risk assessment and mitigation of all ATM System elements throughout the ECAC (European Civil Aviation Conference) States [7]. Although the average trends show a decrease in the number of fatal accidents for Europe, the approach and landing accidents are still the most safety pressing problems facing the aviation industry [32,33,38]. Many relevant repositories[3] report critical incidents involving the ATM System. Unfortunately, even maintaining the same safety levels across the European airspace would be insufficient to accommodate an increasing traffic without affecting the overall safety of the ATM System [6].

The introduction of new safety relevant systems in ATM contexts requires us to understand the risk involved in order to mitigate the impact of possible failures. The EUROCONTROL Safety Regulatory Requirement [8], ESARR4,

[3] Some repositories are: Aviation Safety Reporting Systems - http://asrs.arc.nasa.gov/-; Aviation Safety Network - http://aviation-safety.net/-; Flight Safety Foundation: An International Organization for Everyone Concerned With Safety of Flight - http://www.flightsafety.org/-; Computer-Related Incidents with Commercial Aircraft: A Compendium of Resources, Reports, Research, Discussion and Commentary compiled by Peter B. Ladkin et al. - http://www.rvs.uni-bielefeld.de/publications/Incidents/ -.

requires the use of a risk based-approach in ATM when introducing and/or planning changes to any (ground as well as onboard) part of the ATM System. This concerns the human, procedural and equipment (i.e., hardware or software) elements of the ATM System as well as its environment of operations at any stage of the life cycle of the ATM System. The ESARR4 [8] requires that ATM service providers systematically identify any hazard for any change into the ATM System (parts). Moreover, they have to assess any related risk and identify relevant mitigation actions. In order to provide guidelines for and standardise safety analysis EUROCONTROL has developed the EATMP Safety Assessment Methodology (SAM) [10] reflecting best practices for safety assessment of Air Navigation Systems.

The SAM methodology provides a means of compliance to ESARR4. The SAM methodology describes a generic process for the safety assessment of Air Navigation Systems. The objective of the methodology is to define the means for providing assurance that an Air Navigation System is safe for operational use. The methodology describes a generic process for the safety assessment of Air Navigation Systems. This process consists of three major steps: *Functional Hazard Assessment (FHA)*, *Preliminary System Safety Assessment (PSSA)* and *System Safety Assessment (SSA)*. Figure 1 shows how the SAM methodology contributes towards system assurance.

The process covers the complete lifecycle of an Air Navigation System, from initial system definition, through design, implementation, integration, transfer to operations and maintenance. Although the SAM methodology describes the underlying principles of the safety assessment process, it provides limited information to applying these principles in specific projects. The hazard identification, risk assessment and mitigation processes comprise a determination of the scope, boundaries and interfaces of the constituent part being considered, as well as the identification of the functions that the constituent part is to perform and the environment of operations in which it is intended to operate. This supports the identification and validation of safety requirements on the constituent parts.

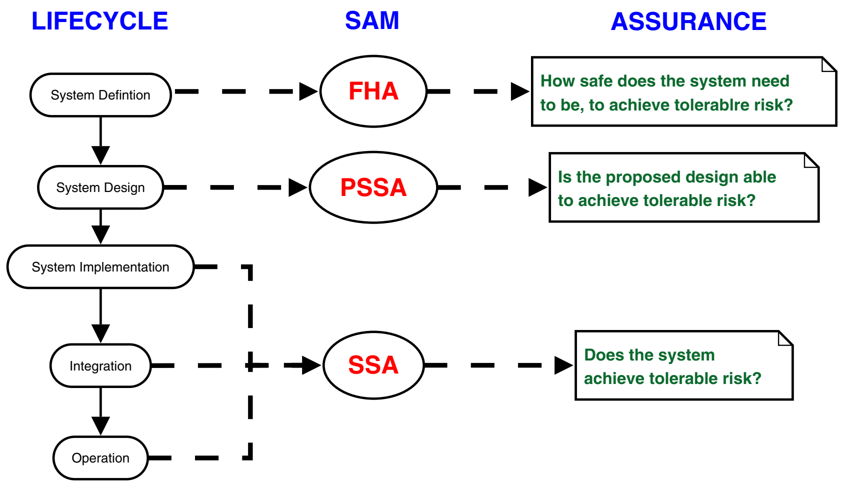

LIFECYCLE
SAM
ASSURANCE
System Defintion
FHA
How safe does the system need to be, to achieve tolerablre risk?
System Design
PSSA
Is the proposed design able to achieve tolerable risk?
System Implementation
Integration
SSA
Does the system achieve tolerable risk?
Operation

Fig. 1. Contribution of the Safety Assessment Methodology towards system assurance

2.1 Limitations

Conventional safety analysis is deemed acceptable in domains such as the nuclear or the chemical sector. Nuclear or chemical plants are well-confined entities with limited predictable interactions with the surroundings. In nuclear and chemical plants design stresses the separation of safety related components from other plant systems. This ensures the independence of failures. Therefore, in these application domains it is possible to identify acceptable tradeoffs between completeness and manageability during the definition and identification of the system under analysis. By contrast, ATM systems operate in open and dynamic environments. Hence, it is difficult to identify the full picture of system interactions in ATM contexts. In particular:

- There is a complex interaction between aircrafts and ATM safety functions. Unfortunately, this complex interaction may give rise to catastrophic failures. Hence, failure independence would increase the overall ATM safety.
- Humans [12,30] using complex language and procedures mediate this interaction. Moreover, most of the final decisions are still demanded to humans whose behaviour is less predictable than that of automated systems. It is necessary further to understand how humans use external artifacts (e.g., tools) to mediate this interaction. This would allow the understanding of how humans adopt technological artifacts and adapt their behaviours in order to accommodate ATM technological evolution. Unfortunately, the evolution of technological systems often corresponds to a decrease in technology trust affecting work practice.
- Work practice and systems evolve rapidly in response to demand and a culture of continuous improvements. A comprehensive account of ATM systems would allow the modeling of evolution. This will enhance strategies for deploying new system configurations or major system upgrades. On the one hand, modeling and understanding system evolution support the engineering of (evolving) ATM systems. On the other hand, modeling and understating system evolution allow the communication of changes across different organisational levels. This would enhance visibility of system evolution as well as trust in transition to operations.

3 Evolutionary Safety Analysis

Capturing cycles of discoveries and exploitations during system design involves the identification of mappings between socio-technical solutions and problems. The proposed framework exploits these mappings in order to construct an evolutionary model that enhances safety analysis. Figure 2 shows the proposed framework, which captures these evolutionary cycles at different levels of abstraction and on diverse models. The framework consists of three different hierarchical layers: *System Modeling Transformation (SMT)*, *Safety Analysis Modeling Transformation (SAMT)* and *Operational Modeling Transformation (OMT)*. The remainder of this section describes the three hierarchical layers.

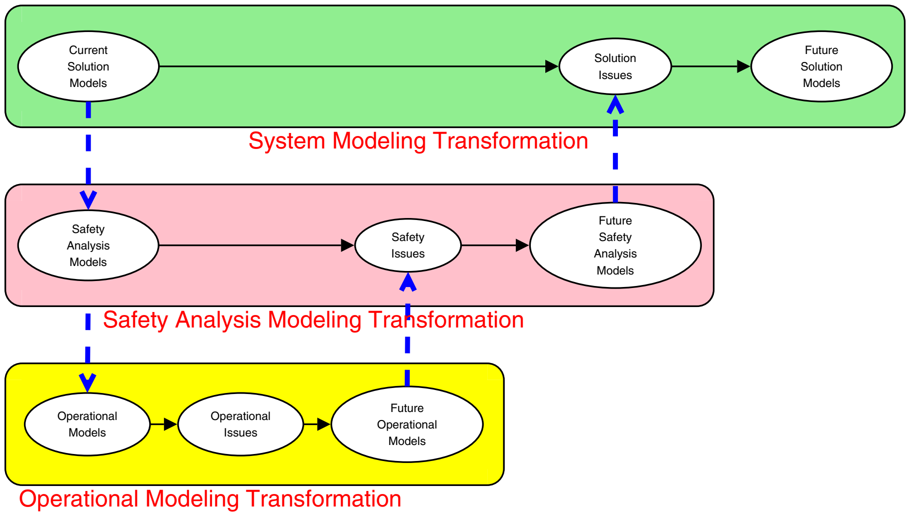

Current Solution Models
Solution Issues
Future Solution Models
System Modeling Transformation
Safety Analysis Models
Safety Issues
Future Safety Analysis Models
Safety Analysis Modeling Transformation
Operational Models
Operational Issues
Future Operational Models
Operational Modeling Transformation

Fig. 2. A framework for modelling evolutionary safety analyses

3.1 System Modeling Transformation

The definition and identification of the system under analysis is extremely critical in the ATM domain. System models used during the design phase provide limited support to safety as well as risk analysis. This is because existing models defined in the design phase are adapted and reused for safety and risk analysis. Organizational and cost-related reasons often determine this choice, without questioning whether models are suitable for the intended use. The main drawback is that design models are tailored to support the work of system designers. Thus, system models capture characteristics that may be of primary importance for design, but irrelevant for safety analysis. Models should be working-tools that, depending on their intended use, ease and support specific activities and cognitive operations of users.

Modeling methodologies and languages advocate different design strategies. Although these strategies support different aspects of software development, they originate in a common *Systems Approach*[4] to solving complex problems and managing complex systems. Modeling incorporates design concepts and formalities into system specifications. This enhances our ability to assess safety

[4] *"Practitioners and proponents embrace a holistic vision. They focus on the interconnections among subsystems and components, taking special note of the interfaces among various parts. What is significant is that system builders include heterogeneous components, such as mechanical, electrical, and organizational parts, in a single system. Organizational parts might be managerial structures, such as a military command, or political entities, such as a government bureau. Organizational components not only interact with technical ones but often reflect their characteristics. For instance, a management organization for presiding over the development of an intercontinental missile system might be divided into divisions that mirror the parts of the missile being designed."*, INTRODUCTION, p. 3, [18].

requirements. For instance, *Software Cost Reduction* (SCR) consists of a set of techniques for designing software systems [14,15]. In order to minimise the impact of changes, separate system modules have to implement those system features that are likely to change. Although module decomposition reduces the cost of system development and maintenance, it provides limited support for system evolution. *Intent Specifications* provide another example of modeling that further supports the analysis and design of evolving systems [25]. In accordance with the notion of semantic coupling, Intent Specifications support strategies (e.g., eliminating tightly coupled mappings) to reduce the cascade effect of changes. Although these strategies support the analysis and design of evolving systems, they provide limited support to understand the evolution of high-level system requirements[5].

Heterogeneous engineering[6] provides a different perspective that further explains the complex interaction between system (specification) and environment. Heterogeneous engineering provides a convenient comprehensive viewpoint for the analysis of the evolution of socio-technical systems. Heterogeneous engineering involves both the systems approach [18] as well as the social shaping of technology [27]. According to heterogeneous engineering, system requirements specify mappings between problem and solution spaces [3,4]. Both spaces are socially constructed and negotiated through sequences of mappings between solution spaces and problem spaces [3,4]. Therefore, system requirements emerge as a set of consecutive solution spaces justified by a problem space of concerns to stakeholders. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings [3,4,11]. The formal extension of these mappings (or solution space transformations) identifies a framework to model and capture evolutionary system features (e.g., requirements evolution, evolutionary dependencies, etc.) [11].

System Modeling Transformation captures how solution models evolve in order to accommodate design issues or evolving requirements. Therefore, an SMT captures system requirements as mappings between socio-technical solutions and problems. This allows the gathering of changes into design solutions. That is, it is possible to identify how changes affect design solution. Moreover, This enables

[5] Leveson in [25] reports the problem caused by Reversals in TCAS (Traffic Alert and Collision Avoidance System): *"About four years later the original TCAS specification was written, experts discovered that it did not adequately cover requirements involving the case where the pilot of an intruder aircraft does not follow his or her TCAS advisory and thus TCAS must change the advisory to its own pilot. This change in basic requirements caused extensive changes in the TCAS design, some of which introduced additional subtle problems and errors that took years to discover and rectify."*

[6] "People had to be engineered, too - persuaded to suspend their doubts, induced to provide resources, trained and motivated to play their parts in a production process unprecedented in its demands. Successfully inventing the technology, turned out to be heterogeneous engineering, the engineering of the social as well as the physical world.", p. 28, [26].

sensitivity analyses of design changes. In particular, this allows the revision of safety requirements and the identification of hazards due to the introduction of a new system. Therefore, the SMT supports the gathering of safety requirements for evolving systems. That is, it supports the main activities occurring during the top-down iterative process FHA in the SAM methodology [10]. The FHA in the SAM methodology then initiates another top-down iterative approach, i.e., the PSSA. Similarly, the framework considers design solutions and safety objectives as input to Safety Analysis. Safety analysis assesses whether the proposed design solution satisfies the identified safety objectives. This phase involves different methodologies (e.g., Fault Tree Analysis, HAZOP, etc.) that produce diverse (system) models. System usage or operational trials may give rise to unforeseen safety issues that invalidate (part of) safety models. In order to take into account these issues, it is necessary to modify safety analysis. Therefore, safety analysis models evolve too.

3.2 Safety Analysis Modeling Transformation

The failure of safety-critical systems highlights safety issues [19,24,31,37]. It is often the case that diverse causes interacted and triggered particular unsafe conditions. Although safety analysis (i.e., safety case) argues system safety, complex interactions, giving rise to failures, expose the limits of safety arguments. Therefore, it is necessary to take into account changes in safety arguments [13]. Figure 3 shows an enhanced safety-case lyfecyle [13].

The lifecycle identifies a general process for the revision of safety cases. Greenwell, Strunk and Knight in [13] motivate the safety-case lifecycle by evolutionary (safety-case) examples drawn from the aviation domain. Figure 4 and 5 show subsequent versions of a safety case. The graphical notation that represents the safety

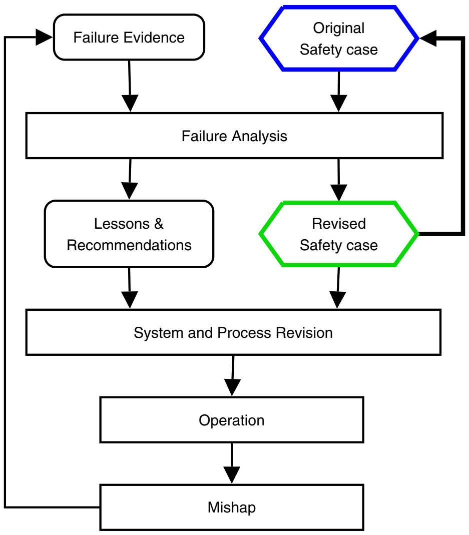


Fig. 3. The Enhanced Safety-Case Lyfecyle [13]

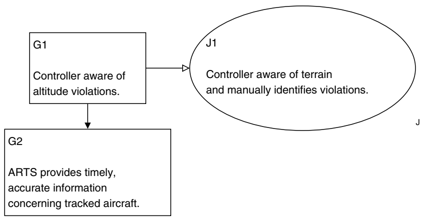

G1
Controller aware of altitude violations.
J1
Controller aware of terrain and manually identifies violations.
J
G2
ARTS provides timely, accurate information concerning tracked aircraft.

Fig. 4. Initial safety argument

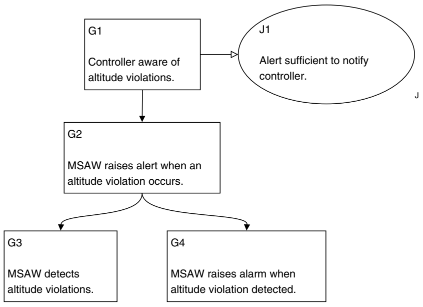

G1
Controller aware of altitude violations.
J1
Alert sufficient to notify controller.
J
G2
MSAW raises alert when an altitude violation occurs.
G3
MSAW detects altitude violations.
G4
MSAW raises alarm when altitude violation detected.

Fig. 5. Revised safety argument

cases is the Goal Structuring Notation (GSN) [21]. Although GSN addresses the maintenance of safety cases, the approach provides limited support with respect to complex dependencies (e.g., external to the safety argument) [20]. Moreover, it lacks any interpretation of the relationships between subsequent safety cases.

Figure 4 shows the initial safety case arguing: *"Controller aware of altitude violations"*. Unfortunately, an accident invalidates the justification J1. The satisfaction of the subgoal G2 is insufficient for the satisfaction of the goal G1. Figure 5 shows the revised safety case that addresses the issue occurred. Unfortunately, another accident, again, invalidates the second safety case [13]. Hence, the safety argument needs further revision in order to address the safety flaw uncovered by the accident.

Figure 6 shows a safety space transformation that captures the safety case changes [11]. The safety case transformation captures the changes from the initial safety case $\mathcal{M}_i^t$ (see, Figure 4) to the revised safety case $\mathcal{M}_i^{t+1}$ (see, Figure 5). An accident invalidates the justification J1. The satisfaction of the subgoal G2 is insufficient for the satisfaction of the goal G1. The proposed safety problem space, $\mathcal{P}_t$, contains these problems, i.e., P_j^t and P_{j+1}^t. The safety space transformation addresses the highlighted problems into the proposed safety case $\mathcal{M}_i^{t+1}$. In order to address the highlighted problems, it is necessary to change the initial safety case. The proposed changes are taken into account in the proposed safety case. Note that there might be different proposed safety cases addressing the proposed safety problem space. The safety space transformation identifies the safety case construction and judgement in terms of safety argumentations and constraints. The safety case consists of the collections of mappings between safety cases and problems. The first part of a safety case consists of the safety argumentations, which capture the relationship that comes from safety cases looking for problems. The second part of a safety case consists of the safety constraints, which capture how future safety cases address given problems. Safety cases at any given time, t, can be represented as the set of all the arcs, that reflect the contextualised connections between the proble space and the current and future safety space. The definition of safety case transformation enables us further to interpret and understand safety case changes, hence safety case evolution [11].

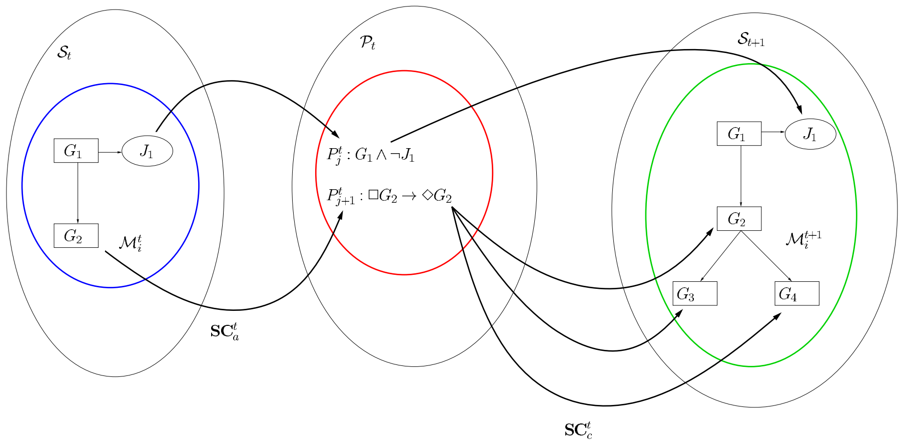


Fig. 6. A safety space transformation

Safety Analysis Modeling Transformation captures how safety analysis models evolve in order to accommodate emerging safety issues. Note that the formal framework is similar to the one that captures SMT. Although design models serve as a basis for safety models, they provide limited supports to capture unforeseen system interactions. Therefore, SAMT supports those activities involved in the PSSA process of the SAM methodology [10]. Note that although the SAM methodology stresses that both FHA and PSSA are iterative process, it provides little supports to manage process iterations as well as system evolution in terms of design solution and safety requirements. The framework supports these evolutionary processes.

3.3 Operational Modeling Transformation

Operational models (e.g., structured scenarios, patterns of interactions, structured procedures, workflows, etc.) capture heterogeneous system dynamics. Unfortunately, operational profiles often change with system usage (in order to integrate different functionalities or to accommodate system failures). Table 1 shows the main problems areas identified in reported incidents: Controller Reports [1] and TCAS II Incidents [2]. Both reports consist of the fifty most recent relevant Aviation Safety Reporting System (ASRS) reports. The small samples are insufficient to identify prevalent issues. However, the two reports highlight the complexity and the coupling within the ATM domain [31]. The analysis of the reports is in agreement with other studies [36,39] that analyse human errors as organizational failures [16,24,34].

Technically, operational observations are reported anomalies (or faults), which may trigger errors eventually resulting in failures. These observations capture *erroneous actions* [16]: *"An erroneous action can be defined as an action which fails to produce the expected result and/or which produces an unwanted consequence"*. In the context of heterogeneous systems (or man-machine systems, or socio-technical systems), erroneous actions usually occur in the interfaces or

Table 1. The main problem areas occuring in two sample incident reports

Problem Areas	Controller Reports	TCAS II Incidents
ATC Facility	2	
ATC Human Performance	44	39
Flight Crew Human Performance	26	40
Cabin Crew Human Performance	1	
Aircraft	3	10
Weather	4	3
Environmental Factor	8	6
Airspace Structure	5	18
Navigational Facility	6	4
Airport	5	5
FAA	3	5
Chart or Publication	1	
Maintenance Human Performance	1	
Company		1

interactions (e.g., man-machine interactions). The cause of erroneous actions can logically lie with either human beings, systems and/or conditions when actions were carried out. Erroneous actions can occur on all system levels and at any stage of the lifecycle.

Capturing operational interactions and procedures allows the analysis of human reliability [16]. In a continuosly changing enviroment like ATM, adaption enhances the coupling between man and machine [17]. Hollnagel in [17] identifies three different adaption strategies: *Adaption Through Design*, *Adaption through Performance* and *Adaption through Management*. Operational Modeling Transformation captures how operational models change in order to accommodate issues arising. The evolution of operation models informs safety analyses of new hazards. Therefore, OMT supports the activities involved in the SSA process of the SAM methodology.

4 Conclusions

This paper is concerned with problems in modeling ATM systems for safety analysis. The future development of ATM, set by the ATM 2000+ Strategy [9], involves a structural revision of ATM processes, a new ATM concept and a systems approach for the ATM network. This requires ATM services to go through significant structural, operational and cultural changes that will contribute towards the ATM 2000+ Strategy. Evolutionary safety analysis captures the judgement of changes. Moreover, it supports the safety assessment of changes from system as well as organisation[7] viewpoints [22,24,34]. Industry (e.g., SMEs) will be involved as early as possible in the lifecycle of ATM projects. The ATM lifecycle

[7] *"Change within an organisation can affect level of safety achieved by that organisation. Change in the institutional structure of an industry can affect the level of safety achieved by the industry as a whole."*, p. 6, [22].

involves various stakeholders (e.g., Institutional, Solution Providers, Society and Other Industries) [22] assuming different roles with respect to safety judgement. Unclear responsibilities and ownerships, with respect to safety cases, affect the trustworthiness of safety analysis [22]. Evolutionary safety analysis, therefore, requires the identification of responsibilities and ownerships in order to address institutional issues (e.g., institutional changes, inappropriate ownerships, etc.).

In conclusion, this paper introduces a framework that supports evolutionary safety analysis. Although existing processes emphasise the iterative nature of safety analysis, they provide limited support to capture evolutionary transformations. The framework captures evolutionary safety analysis. Examples drawn from the ATM domain show the different relationships between subsequent evolutionary models. The systematic production of safety analysis (models) will decrease the cost of conducting safety analysis by supporting reuse in future ATM projects.

Acknowledgements. This work has been supported by the UK EPSRC Interdisciplinary Research Collaboration in Dependability, DIRC - http://www.dirc.org.uk - grant GR/N13999.

References

1. Aviation Safety Reporting System. *Controller Reports*, 2003.
2. Aviation Safety Reporting System. *TCAS II Incidents*, 2004.
3. Mark Bergman, John Leslie King, and Kalle Lyytinen. Large-scale requirements analysis as heterogeneous engineering. *Social Thinking - Software Practice*, pages 357–386, 2002.
4. Mark Bergman, John Leslie King, and Kalle Lyytinen. Large-scale requirements analysis revisited: The need for understanding the political ecology of requirements engineering. *Requirements Engineering*, 7(3):152–171, 2002.
5. BFU. *Investigation Report, AX001-1-2/02*, 2002.
6. John H. Enders, Robert S. Dodd, and Frank Fickeisen. Continuing airworthiness risk evaluation (CARE): An exploratory study. *Flight Safety Digest*, 18(9-10):1–51, September-October 1999.
7. EUROCONTROL. *EUROCONTROL Airspace Strategy for the ECAC States, ASM.ET1.ST03.4000-EAS-01-00*, 1.0 edition, 2001.
8. EUROCONTROL. *EUROCONTROL Safety Regulatory Requirements (ESARR). ESARR 4 - Risk Assessment and Mitigation in ATM*, 1.0 edition, 2001.
9. EUROCONTROL. *EUROCONTROL Air Traffic Management Strategy for the years 2000+*, 2003.
10. EUROCONTROL. *EUROCONTROL Air Navigation System Safety Assessment Methodology*, 2.0 edition, 2004.
11. Massimo Felici. *Observational Models of Requirements Evolution.* PhD thesis, Laboratory for Foundations of Computer Science, School of Informatics, The University of Edinburgh, 2004.
12. Flight Safety Fundation. *The Human Factors Inplication for Flight Safety of Recent Developments In the Airline Industry*, number (22)3-4 in Flight Safety Digest, March-April 2003.

13. William S. Greenwell, Elisabeth A. Strunk, and John C. Knight. Failure analysis and the safety-case lifecycle. In *Proceedings of the IFIP Working Conference on Human Error, Safety and System Development (HESSD)*, pages 163–176, 2004.
14. Constance L. Heitmeyer. Software cost reduction. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Waley & Sons, 2nd edition, 2002.
15. Daniel M. Hoffman and David M. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2001.
16. Erik Hollnagel. *Human Reliability Analysis: Context and Control*. Academic Press, 1993.
17. Erik Hollnagel. The art of efficient man-machine interaction: Improving the coupling between man and machine. In *Expertise and Technology: Cognition & Human-Computer Cooperation*, pages 229–241. Lawrence Erlbaum Associates, 1995.
18. Agatha C. Hughes and Thomas P. Hughes, editors. *Systems, Experts, and Computers: The Systems Approach in Management and Engineering, World War II and After*. The MIT Press, 2000.
19. Chris W. Johnson. *Failure in Safety-Critical Systems: A Handbook of Accident and Incident Reporting*. University of Glasgow Press, Glasgow, Scotland, October 2003.
20. T. P. Kelly and J. A. McDermid. A systematic approach to safety case maintenance. In Massimo Felici, Karama Kanoun, and Alberto Pasquini, editors, *Proceedings of the 18th International Conference on Computer Safety, Reliability and Security, SAFECOMP'99*, number 1698 in LNCS, pages 13–26. Springer-Verlag, 1999.
21. Timothy Patrik Kelly. *Arguing Safety - A Systematic Approach to Managing Safety Cases*. PhD thesis, Department of Computer Science, University of York, 1998.
22. Steve Kinnersly. Whole airspace atm system safety case - preliminary study. Technical Report AEAT LD76008/2 Issue 1, AEA Technology, 2001.
23. Jean-Claude Laprie et al. Dependability handbook. Technical Report LAAS Report no 98-346, LIS LAAS-CNRS, August 1998.
24. Nancy G. Leveson. *SAFEWARE: System Safety and Computers*. Addison-Wesley, 1995.
25. Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1):15–35, January 2000.
26. Donald A. MacKenzie. *Inventing Accuracy: A Historical Sociology of Nuclear Missile Guidance*. The MIT Press, 1990.
27. Donald A. MacKenzie and Judy Wajcman, editors. *The Social Shaping of Technology*. Open University Press, 2nd edition, 1999.
28. Stuart Matthews. Future developments and challenges in aviation safety. *Flight Safety Digest*, 21(11):1–12, November 2002.
29. Michael Overall. New pressures on aviation safety challenge safety management systems. *Flight Safety Digest*, 14(3):1–6, March 1995.
30. Alberto Pasquini and Simone Pozzi. Evaluation of air traffic management procedures - safety assessment in an experimental environment. *Reliability Engineering & System Safety*, 89(1):105–117, July 2005.
31. Charles Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press, 1999.
32. Harro Ranter. Airliner accident statistics 2002: Statistical summary of fatal multi-engine airliner accidents in 2002. Technical report, Aviation Safety Network, January 2003.

33. Harro Ranter. Airliner accident statistics 2003: Statistical summary of fatal multi-engine airliner accidents in 2003. Technical report, Aviation Safety Network, January 2004.
34. James Reason. *Managing the Risks of Organizational Accidents.* Ashgate Publishing Limited, 1997.
35. Review. Working towards a fully interoperable system: The EUROCONTROL overall ATM/CNS target architecture project (OATA). *Skyway*, 32:46–47, Spring 2004.
36. Scott A. Shappell and Douglas A. Wiegmann. The human factors analysis and classification system - HFACS. Technical Report DOT/FAA/AM-00/7, FAA, February 2000.
37. Neil Storey. *Safety-Critical Computer Systems.* Addison-Wesley, 1996.
38. Gerard W.H. van Es. A review of civil aviation accidents - air traffic management related accident: 1980-1999. In *Proceedings of the 4th International Air Traffic Management R&D Seminar*, New-Mexico, December 2001.
39. Douglas A. Wiegmann and Scott A. Shappell. A human error analysis of commercial aviation accidents using the human factors analysis and classification system (HFACS). Technical Report DOT/FAA/AM-01/3, FAA, February 2001.

Public-Key Cryptography and Availability*

Tage Stabell-Kulø and Simone Lupetti

Department of Computer Science, University of Tromsø, Norway
pesto@pasta.cs.uit.no

Abstract. When the safety community designs their systems to also maintain security properties, it is likely that public-key encryption will be among the tools that are applied.

The security guarantees of this technology are based on a particular model of computation. We present the properties of this model that are relevant in the setting of distributed systems. Of particular importance is that the model has no notion of time.

From this it follows that systems that need to be available must exercise the utmost care before applying public-key encryption in any form. We discuss the relation between public-key encryption and timeliness, the tradeoffs that must be made at design time, and how the property of (lack of) availability might very well contaminate other system components.

1 Introduction

It is reasonable to expect that the safety community will be forced to deal with an ever-increasing number of security issues (as opposed to only those related to safety) [1]. To solve security problems it is likely that technology from the security community will be applied. Shared-key and public-key encryption, message digests, and digital signatures are all based on advanced mathematical concepts, and these mechanisms are routinely used in security engineering [2].

As is probably the case in other communities, the world-view of the security community is rarely discussed with outsiders. Among insiders this view is more or less taken for granted, and when presenting to outsiders the technology that has been developed there is no room for lengthy depositions on the inner workings of the computational model.

In the world-view of the security community, the most important goal is often, simply said, the prevention of "bad things" happening in and to the system. This is often considered so important that it is permitted to achieve such prevention in ways that also now and then prevent "good things" from happening. After all, it is better to be secure than sorry, and anything less than perfect security is sometimes taken to be equivalent to no security [3]. *Availability* is one of these "good things", and even though availability certainly is a security property,

* This work has been generously supported by the Research Council of Norway by means of the Arctic Bean project (IKT 2010, project number 146986/431) and the Penne project (IKTSoS project number 158569/431).

more often than not, security is only concerned with upholding other security properties such as non-repudiation, integrity and confidentiality.

To shed light on some of the implications of security technology we will use the approach of negative requirements [4]. As our example of a sophisticated security technology that may negatively affect critical system properties, we take public-key encryption [5]. Instead of presenting how public-key encryption can for example be used to turn a channel with integrity into one with secrecy without the exchange of secrets, we focus on what public-key encryption *cannot do*. By demonstrating what properties can not be achieved with public-key encryption, we can say that in systems where these properties are important, public-key encryption should not be used, or at least great care should be taken in the way it is applied.

It is intrinsic to public-key encryption to block in certain situations (and hence become unavailable) [6]. Thus, any system that relies on public-key encryption will be prone to blocking. In essence: The blocking behavior (and thus the whole issue of unavailability) is closely related to certain security properties of public-key encryption. The problem is that it can be impossible in advance (at design time) to determine under which conditions blocking will occur. Furthermore, when public-key encryption is used in the system, it contaminates other components with its blocking behavior. The core of the issue is that if a system for safety reasons can not tolerate blocking, it can not tolerate to be contaminated by the blocking property of public-key encryption.

The outline of the rest of the paper is as follows. In Section 2 we present terminology, and in Section 3 the model of computation used in the security community in sufficient detail to enable us to facilitate a precise discussion. Then, in Section 4 we discuss properties of public-key encryption with focus on what can not be facilitated. Some alternatives are discussed in Secion 5. We conclude in Section 6.

2 Terminology

Our terminology follows [7]; Please notice that it is slightly different from that traditionally used in engineering.

Error: A design flaw. This can be a specification that is outright wrong, or a lack of specification;
Availability: Readiness for correct service;
Reliability: Continuity of correct service;
Failure: The nonperformance or inability of the system or a single component to performs its intended function;

Note that a failure is defined as an event, while an error is a static condition. This means that a failure *occurs*, while an error *remains* in that it is part of the system until it is removed (usually by human intervention). A design can have provisions to counter failures, and those that are properly dealt with obviously have no further consequences. Not doing so is an error.

We further define a *system-wide failure* to be one where the system is not only unable to operate, but where an extra-system intervention is required. For example, if the power supply in a machine fails, no program on the stopped machine is able to act and we have a system-wide failure (for this machine).

Denial of Service (DOS) is an attack on (parts of) the system to interrupt it's normal mode of operation. Such an attack can take place as part of normal operation (flooding a server with requests, for example), or as an extra-system attack (blocking the flow of water to the air condition to ensure that machines shut down due to over heating). The purpose is often of a secondary nature: By interrupting one part of the system, some other parts are also negatively affected. Well designed and engineered components should be able to recover from a DOS attack, but if the purpose is simply to deny service (for a period of time), recovery does not help on the original problem (which is denial of service) [8,9].

3 Models for Computer Security

The computer security community has two models of computation that are relevant to us: The one used to represent a distributed system, and the one describing encryption. We now discuss them in turn.

3.1 Distributed Systems

The design and implementation of public-key systems rest firmly on the model that the computer security community use to describe distributed systems. It is possible to describe most of the functional aspects of distributed systems using this model. In order to understand the semantics of public-key technology it is prudent to first discuss the model on which it is supposed to rest.

The fundamental abstractions of computer security is that the system consists of a set of *processes*. Processes interact in order to create and keep a *shared state* but also maintain an internal state, of which we can know nothing except by asking them and analyzing the result.

A process can (and many times it does) fail: there are countless ways in which this can happen. A comfortable assumption is that processes are either running, or they are not; this is called *crash fail* [10]. The idea is that a process that has failed remains in a blocked state, but, upon restart, is able to take the actions necessary to restore (its part of) the shared state. If security is important at all, this model is probably too simplistic and processes are instead assumed to potentially fail also by being malicious; in this case we have a so-called Byzantine (or arbitrary) failure [11]. Notice that assuming Byzantine failure also encompasses insiders that for some reason try to harm the system. In fact, this is probably the most common source of Byzantine failures. Evidently wrong (meaningless) data generated by random failures (often called glitches) will in most cases be detected by checksums and other mechanism, and automatically discarded. Apparently legal (meaningful) data forged by an attacker can on the other hand be accepted as genuine unless provisions have been taken. Byzantine failure describes, for example, the kind of failures that we can expect in presence of Trojan and viruses and in all other cases where authorization policies are circumvented.

Processes run logically separated from each other, and can communicate solely by exchanging messages. Messages travel, independently, on communication links. No interesting assumptions are usually made about these links. In particular, it is not assumed that a message actually manages to traverse the link, that it does so without being corrupted (with or without malice), that messages sent in order retain their order upon arrival, or that no duplication takes place. Notice that not making any of these assumptions is consistent with the failure model of the Internet. In particular, links can fail completely by dropping all messages.

Links that have failed completely may create *partitions*. In general it is impossible to know how many processes reside in a partitioned portion of a distributed system and for how long the partition will last. The (lack of) assumptions about communications implies also that it is impossible to know the difference between a crashed process and a link that has failed completely. For the same reason it is impossible to distinguish between a process that is acting in a Byzantine way and a link that is modifying messages on the fly. Since interaction with a Byzantine process is not fruitful, Byzantine failures can also be viewed as a failure that creates a partition in the system.

The main implication of this is that, in this model, failures always manifest themselves as communication problems. Any communication problems may create (possibly temporary) partitions. This has profound effects on the notion of progress: is the system constructed in such a way that progress is possible when one (or more) nodes are left out (but not known to have stopped or crashed)? What availability means in the specific system and how it is related to partitions can only be answered by looking at the policies of the system (and, thus, can not be answered in general).

For example, if progress of individual users is the goal of the service offered by the system, it would not be wise to design the system in such a way that a partition can stop a user from carrying on with his work [12]. Otherwise, some sort of majority of processes taking responsibility for the global computation (enforcing availability) could be found even if this implies to arbitrarily make decisions about other processes' state. In technical terms this last problem requires the establishment of *consensus* that in turn can not be solved without blocking for an unbounded length of time in the types of systems we consider [13].

The upshot of all of this is that there are two interesting objects to consider in a distributed system, namely links and processes. Furthermore, there are two modes of failure that concerns us, failure by crashing and Byzantine failure. Both types, regardless of whether it happens in a process or in a link, may lead to partitions. Partitions, by definition, are availability problems.

3.2 Encryption

We will assume "perfect encryption" [14]. We make this sweeping assumption in order to confine the discussion to other aspects of public-key encryption than the encryption itself. Perfect encryption only makes sense under a certain technological regime. Perfect encryptions emcompass the assumption that no fundamental change in the traditional computational model takes place during the lifetime of the system, such as if quantum cryptography becomes available, it is shown that $P = NP$, or some other fundamenal property on which encryption rests is changed. In this case, our assumption on perfect encryption does not hold and all results must be examined again.

4 Public-Key Encryption

In this section we will discuss how and why public-key encryption gives rise to a choice between availability and security.

Security mechanisms are designed under the assumption that system components maintain their integrity. On one hand this lets us design sophisticated run-time mechanisms that both provides security and high availability, on the other hand practice has shown that alteration of the system state is the first goal of potential attacks. Since it is impossible to guarantee system integrity (for all its possible definitions) it is often vital to have a clear view of the system's *failure model.*

As our running example we will discuss one of the many possible disruptions in a system where public-key encryption is in use: That of a key being compromized. There are many ways a secret key can leak out, ranging from malice to negligence. In any case, if a key has become known (or it is feared that a key has become known) the key must immediately be *revoked.* The revocation demands that any holder of the key should cease using it without delay, and in general we can expect a revocation to signal a serious security incident. Notice that we are not concerned with the continuous revoking of old keys that is part of the normal operation of the system, which in itself introduces a wide range for engineering issues.

We will discuss several aspects of revocation, but in any case, a solution to the issue of revocation is an intrinsic part of the security properties of public key encryption. A solution must be designed for each and every system even though revocation has nothing to do with the cryptographic properties as such.

The following are well-known problems arising from the application of public-key encryption (see for example [6]), but we have cast them in such a way that the tradeoff between security and availability becomes evident.

4.1 Who Can Revoke a Key

It must be known in advance who is authorized to revoke a key.

Obviously, a malicious (or erroneously) revocation of some (or all!) of the keys in the system will most likely be a system-wide failure. It is impossible to arrange things so that this can not happen (if keys can be revoked at all), but one can make it as unlikely as one desires. For example, by means of certificates we can create a *compound principal* such as "Alice **and** Bob **as** Revoke Authority"[15]. When this regime is in place only Alice *and* Bob (in concert) can revoke a key, and neither Alice nor Bob can revoke keys alone. However, revoking a key now requires both Alice and Bob to be available, and this creates a problem of reliability. In concrete terms, from a security point of view there is now a single point of failure in the system: A successful DOS against either Alice or Bob (or both) will paralyze the authority to revoke. In fact, any partition between Alice and Bob will have this effect, regardless of how it comes about.

Because the principal having authority to revoke keys is very powerful, the mechanisms put in place to control it should involve as many participants as possible to guard against malicious attacks, while at the same time as few as possible to ensure that a key can be revoked without delay.

From this we see that designing and implementing a policy for management of authority to revoke keys involves mainly system-specific issues. The design needs to take into consideration the general threat model of the system, the potential costs of not revoking in a timely fashion, the reliability of the network as a whole, the probability of a malicious entity revoking keys, and a host of other issues. Most of these can not be calculated, and estimates must be made on which to base the decision (be means of simulations, for example).

The design of the mechanisms that are to guard the authority to revoke keys is an exercise in the tradeoff between security on one hand and availability on the other.

4.2 How to Distribute a New Key

After a key has been revoked, a new key must be distributed in some predetermined manner.

Assume that Charlie's key has been revoked. Until a new key has been disseminated, Charlie is effectively silenced. No one will be able to send him data without violating system security, and data coming from him will be discarded for the same reason. Or, in other words, the part of the system controlled by Charlie is disconnected and so unavailable. The need for security was deemed higher than the need for availability.

One could lump together the authority to create new keys (and certify them) with the authority to revoke keys, but there is no need to do so. In fact, for reasons of security, you probably should not do so. The problem is that on the one hand the message revoking the key should be spread as fast as possible while on the other hand, (parts of) the system might be paralyzed before a new key can be installed. The window can obviously be made to be zero by always issuing the new key together with the certificate that revokes the old one, but this again requires a co-location of the authority that revokes and the one that "restarts" the system.

It is most likely a system-wide failure if the (possibly combined) principal that issues new keys fails by issuing unwarranted keys. As usual, one can make the reliability of this service as high as one deems necessary at the cost of availability.

4.3 How to Spread the Revocation

The notification that a key has been revoked must be spread to all those that potentially hold the key. One can assume that a key will not be revoked unless there is a reasonable strong belief that the key constitutes a security problem. That is, we can assume that the time from which the key becomes known and until all participants has received the message to revoke the key constitutes a window of vulnerability. The problem is that the nature of the task at hand makes it possible for an attacker to make this window of vulnerability as long as he wants. We will examine this issue below.

There are two means of spreading information (a revocation in this case) in a distributed system: Either the information is pushed, or it is pulled [15].

Pushing the information is the simplest solution in that a message is sent to all participants. However, there is no way of knowing that all participants actually receive the message, and if the number of participants is large and their physical distance great, the probability of success of this approach will be rather low. The alternative, to engage in some protocol, is equivalent to creating consensus. Such protocols can be blocking, and are at best probabilistic, where the probability is a function of the characteristics of the physical network (over which processes do not have control). In this state the system is particularly vulnerable to denial of service attacks as security has been breached and the window is open as long as messages are hindered. In other words, pushing is not very secure.

The alternative to pushing is pulling. Each key is augmented with a certificate that requires the one using it to verify that the key is still valid; the details of such an on-line service for verification can be found in [15]. The problem is that in this case the user is blocked if he can not reach the verification service. Again, this service can be made as reliable as one wishes, at the cost of lowering security (the more servers to update in case of a revocation the longer the window of vulnerability).

Another tradeoff is to use a somewhat less reliable but more secure verification service, but issue the verification certificates with a lifetime. But, again, how long this timeout is, will again be a tradeoff between availability and security that needs to be determined in advance.

4.4 Recovery from a Leaked Key

Assume that the principal authorized to revoke a key has decided that based on the available information, a certain key must be revoked. In many cases this only happens after the fact; it becomes known that at some time in the past some event occurred that endangered (the secret part) of a public key. Let us denote the time at which it is decided that the compromise occurred with t.

The compromise has two implications: Messages encrypted with the public key after time t can no longer be assumed to be secret, and signatures made with the key after time t can no longer be assumed to be authentic without scrutinizing of the events leading up to where the signature being made.

If loss of secrecy and/or authenticity is a system-wide failure, a strategy for recovery must be in place. This strategy will determine who has authority to revoke the key, how to spread the revocation, but also how to deal with all messages encrypted with the key since time t. This recovery procedure can be utterly complicated, and while it is in progress the system might be very vulnerable against DOS attacks, among other things.

To design and implement such a recovery mechanism, while maintaining all other properties of the system, will require a sage tradeoff between security and availability.

4.5 Public-Key Infrastructure (PKI)

We have on purpose avoided the term PKI in our disposition. In the above examples it is evident that *some* means must be found to disseminate keys and certificates, offer verification (and thus revocation) services, to coordinate the activities of the certification authorities, and so on. We believe that whether this is organized under the umbrella of a PKI, or in some other way does not alter any of the arguments we have presented.

4.6 Summary

We see from the examples above that they all reveal the need for a tradeoff between security and availability. Although there are cases where one must surely be selected before the other, this is in general a difficult task. In particular, in most cases it is of prime importance to lower as far as possible the probability of any system-wide failure. The problem is that even though it is obvious that the system as a whole has a certain probability of failure, actually finding it might not be feasible. In particular, due to the complexity of the system, the actual tradeoff that has been done will often not be visible before recovery is necessary.

We believe that the examples we have shown demonstrates that including public-key encryption in a system gives rise to a large set of issues that must be addressed, and that all of them hinge on the probabilities of (a set of) events to occur (or not occur).

In addition to the issues discussed here, public-key encryption introduces also other security properties that need to be considered. For example, the holder of a public key can anonymously send encrypted messages, and the presence of public-key encryption gives rise to the need for authentication. Moreover, it is impossible to know who holds a public key, and thus to know who will verify signatures in the futures and possibly use the signature for a malign purpose.

5 Alternatives

In a system without full physical control over the communications links, encryption is the only means available to ensure authentication (and thus authorization) and integrity. There are three technologies that are readily at hand: Symmetric key (e.g., DES), asymmetric key (public key, e.g., RSA), and hashing (digital fingerprint, e.g., SHA). The challenge is to use them in the most convenient manner.

In general we can say that public-key encryption excels in systems where the participants have no prior knowledge of each others. This is seldom the case, electronic commerce with the general public aside. But most of the abstraction it offers can be also obtained using other technologies. As an example, let us demonstrate how to obtain digital signatures using shared keys only.

Assume the three parties Alice, Bob and a trusted Server. Assume furthermore that rather than being trusted to realize a PKI, S shares a key with A (K_{AS}) and one with B (K_{BS}), and that A and B has exchanged a session key (K_{AB}) by some means (for example in concert with establishing the Service Level Agreement). The message

$$A \rightarrow B : \{\{M\}_{K_{AS}}, M\}_{K_{AB}}$$

gives Bob all the evidence he needs to hold M against Alice, with the assistance of S. This places the same responsibilities on S as would the combination of implementing a PKI and a CA. Notice that in both the solution for shared-key and public-key encryption S must be trusted to be willing and able to do "the right thing". If S fails to be trustworthy, both technologies fail to provide a solution. The only difference is precisely what this "the right thing" is. Or, in other words: Establishing digital signatures is a matter of establishing and maintaining trust rather than of cryptographic technology.

Another issue for concern is the ability to keep keys secret. In public-key encryption there is also a key-component that must be kept secret, and the engineering challenges are not smaller for this technology than for shared-key encryption; keeping one key secret is not much more complicated than keeping thousand keys secret. Also in this respect the tradeoff is more biased by trust and belief in the ability of participants to uphold local security policies, than of technology.

Taken together we can say that whether the management necessary to support shared-key encryption is a heavier burden to carry than that of public-key encryption is system dependent. Providing universally valid guidelines is probably impossible.

6 Conclusions

The security community has an array of powerful technologies to offer systems designers. We have discussed but one: Public-key encryption for secrecy (encrypting with the public key) and authentication (digital signatures).

Failures are inherit in the computational model of distributed systems, and this creates problems. To uphold security properties, public-key encryption must be supported by complex and distributed infrastructure. Taken together, when public-key encryption is used, a tradeoff must be found between availability on one hand, and security on the other. The availability of the system is heavily affected by this decision.

Public-key encryption introduces many lanes that can lead to system-wide failures, and that can lead to blocking (denial of service) in whole or parts of the system. Unfortunately, there does not seem to be any structured manner in which to proceed, as all tradeoffs must be made based on the actual network topology, the properties of the resources that must be protected, and so on. In particular, it seems as if the tradeoffs *must* be made at the time of deployment.

All of this should be contrasted to the use of symmetric keys. The process of exchanging keys can be complex, and shared keys must be protected; just as the secret part of in the key-pair in a public-key system. But with a shared key it is clear with whom you share a key, it is clear who can authenticate you, anonymous receiver and senders are not feasible, and so on.

The lesson to be learned is that although the somewhat troublesome properties of public-key encryption is well known in the security community, this might not be the case in the safety community. From this it should follow that such powerful abstractions as digital signatures should only be applied if they are fully understood. In particular, if blocking in any form is a problem in the target system, authentication and integrity should be achieved by other means than by using public-key encryption.

This does not necessarily mean to avoid public-key technologies in all cases but to carefully examine all the possibilities not letting its recognized power to mask its drawbacks: When valid alternatives are available the choice can not be obvious.

Acknowledgments

This paper would not have been written had it not been for the IKTSoS workshop arranged by the Norwegian Research Council in March 2005. Feico Dillema, Jon Ølnes, Arne Helme and Dmitri Zagorodnov acted as catalyzators in crystalizing our ideas.

References

1. Pfitzmann, A.: Why safety and security should and will merge. In Heisel, M., Liggesmeyer, P., Wittmann, S., eds.: Proceedings of Computer Safety, Reliability, and Security (SAFECOMP'04). Volume 3219 of Lecture Notes in Computer Science., Potsdam, Germany, Springer (2004) 1–2
2. Anderson, R.J.: Security Engineering. John Wiley & Sons, Inc. (2001)
3. Lampson, B.: Security in the real world. IEEE Computer **37** (2004) 37–46
4. Rushby, J.: Critical system properties: Survey and taxonomy. Reliability Engineering and System Safety **43** (1994) 189–219
5. Nechvatal, J.: Public key cryptography. In Simmons, G.J., ed.: Contemporary cryptology, the science of information integrity. IEEE Press (1992) 177–288
6. Roe, M.: Cryptography and evidence. PhD thesis, Clare College, University of Cambridge, UK (1998)
7. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1** (2004) 11–33
8. Needham, R.M.: Denial of service: an example. Communications of the ACM **37** (1994) 42–46
9. Mirkovic, J., Reiher, P.: A taxonomy of DDoS attack and DDoS defense mechanisms. SIGCOMM Computer Communication Review **34** (2004) 39–53

10. Barborak, M., Dahbura, A., Malek, M.: The consensus problem in fault-tolerant computing. ACM Comput. Surv. **25** (1993) 171–220
11. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4** (1982) 382–401
12. Stabell-Kulø, T., Dillema, F., Fallmyr, T.: The open-end argument for private computing. In Gellersen, H.W., ed.: Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing. Number 1707 in Lecture Notes in Computer Science, Springer Verlag (1999) 124–136
13. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
14. Blum, M., Goldwasser, S.: An efficient probabilistic public-key encryption scheme which hides all partial information. In: Proceedings of Advances in Cryptology—Crypto'84. Volume 196 of Lecture Noets in Computer Science., Springer verlag (1984)
15. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distribued systems: theory and practice. ACM Transactions on Computer Systems **10** (1992) 265–310

End-To-End Worst-Case Response Time Analysis for Hard Real-Time Distributed Systems

Lei Wang[1, 2], Mingde Zhao[1], Zengwei Zheng[1, 3], and Zhaohui Wu[1]

[1] College of Computing Science, Zhejiang University, 310027 Hangzhou, P. R. China
{alwaysbeing, zmdd48, zhengzw, wzh}@cs.zju.edu.cn
[2] UFSOFT School of Software, Jiangxi University of Finance & Economics, 330013 Nanchang, R. P. China
alwaysbeing@sohu.com
[3] City College, Zhejiang University, 310015 Hangzhou, P. R. China
zhengzw@zucc.edu.cn

Abstract. The verification of end-to-end response time for distributed hard real-time systems is quite necessary for safety-critical applications. Although current time analysis techniques can precisely analyze the response time, the performance is not quite satisfying, especially for large size distributed systems. This paper presents a novel end-to-end worst-case response time analysis approach for hard real-time distributed systems. This technique is based on the critical instant analysis and the canonical form transformation. It extends the traditional holistic analysis technique by exploiting the precedence relations between tasks on both different processors and the same processor. Simulation results have shown that this algorithm can achieve accurate results and offers good performance for systems with wide range of CPU utilizations and task set size, and therefore is applicable to schedulability analysis of complex distributed systems.

1 Introduction

Distributed control systems are increasingly deployed in current industry and products, such as manufacturing plant, aircraft and vehicles, in which, many control systems are decentralized at different locations and interconnected through one or more buses or networks, over which data and control signals are transmitted. The distributed architecture is superior to the traditional centralized control systems in performance, capability and robustness, et al. The software in these systems is composed of concurrent tasks that are often statically allocated to processing nodes, and may exchange messages with other tasks on the same node or on different nodes. Control algorithms are usually designed and implemented with the assumption of a periodic behavior, which is often important in achieving the required control performance and stability. There are strict deadlines placed on the response times of control processes. Failure to meet the deadline may result in catastrophic consequences. In order to verify that a given control system is capable of providing the required quality of control, a means to predict end-to-end response times is required. Preferably, this information should be available at an early design stage.

Schedulability analysis of fixed priority preemptive tasks known as RMA (Rate Monotonic Analysis) has been well developed since Liu and Layland's seminal paper in 1973 [1]. The set of analysis methods now handles a wide range of systems in which the original restricting assumptions of RMA have been successively removed. Systems that can be analyzed include systems with task synchronization [2], a mixture of periodic and aperiodic tasks [3] [4], tasks with arbitrary deadlines [5], etc.

Even though RMA was initially formulated as a theory for analysis of tasks executing on a single processor, it is also extended to cover scheduling analysis in distributed systems. An approach to calculate end-to-end response times for distributed real-time systems is the holistic scheduling analysis proposed by Klein [6] and Tindell [7]. It makes all the tasks independent by introducing a jitter on release time due to the execution delay of preceding tasks and message delivery and computes worst-case response times for transactions by summing individual task and message response times. However, when precedence relations take place on the same processors, the holistic analysis may be pessimistic since it does not take into account this situation. To decrease the pessimism, Palencia et al. introduced offsets [8] and best-case response times [9] into the schedulability analysis. In [10] they extended the technique in order to exploit more accurately the precedence relations on the same processors. However, the algorithm based on this technique is time-consuming and can hardly obtain an accurate result within a human-tolerable period of time when analyzing complex distributed systems.

In this paper, we extend the holistic analysis technique to deal with the precedence relations between tasks in the same processors. The analysis technique is based on the traditional critical instant analysis and the canonical form transformation proposed by Harbour et al. [11], in which tasks' time behaviors are accurately calculated. Compared with the current techniques, this algorithm offers higher performance for systems with wide range of CPU utilizations and task set size on the condition that the accuracy is guaranteed, therefore it is applicable to the schedulability analysis of complex distributed real-time systems.

The remainder of the paper is organized as follows: Section 2 describes the assumptions and computational model used in this paper. Section 3 proposes the methods to compute end-to-end response times and release jitters. Then an extended worst-case response time analyses algorithm and an integrated schedulability analysis algorithm are presented in Section 4. Simulations result is given in Section 5. Finally, we conclude the paper in Section 6.

2 Assumptions and Computational Model

In this paper we consider event-driven software systems. In these systems a set of external event sequences activate tasks. These tasks may generate internal events (signals as well as data) that activate other tasks, and so on. We assume that a task immediately activates its successors at the end of its execution. All tasks, which are in precedence relation by their activation, are grouped into entities that we called transactions. The assignment of the tasks to the processors is given and all the tasks of the same transaction do not necessary assigned to the same processor. Each transaction has a period and deadline. We assume the deadline is less than or equal to their period. Tasks within the same transaction have the same period and deadline with the transaction. We exclude the possibility that the execution of multiple instances of a task can be underway, i.e., a task must complete before its next arrival. In the computational model referenced in this paper, each task has one direct predecessor at most, which may run on the same or different processor. A task may have one or more successors on the same or different processors.

On each processor, tasks are statically assigned unique priority and scheduled under a fixed priority preemptive strategy. The assignment of priority is not the focus in this paper, but we assume that predecessors are always assigned higher priority than their successors on the same processor. This is reasonable since successors in a transaction are always released after their predecessors complete. To reduce the complexity of response time analysis, the communication costs between processors is neglected in this study. However, the response time of messages can be analyzed with the similar method by modeling each network as if it were a processor, and each message as if it were a task [12].

Let P be the set of processors. There is a set $\Gamma = \{t_1,\ldots, t_n\}$ of n periodic tasks in the system. Each task t_i is characterized by $(T_i, C^W_i, D_i, \pi_i, p_i)$ where T_i is the period of t_i, C^W_i are its worst-case execution times (WCET) respectively, D_i is its deadline, π_i is its priority and $p_i \in P$ is the processor to which it is allocated. Γ_p denotes the task set allocated to the processor $p \in P$.

Let k be the total number of transactions in the system. There is a set $X = \{x_1,\ldots, x_k\}$ of k transactions in the system. Each transaction x_i is characterized by (T_i, TS_i, D_i) where T_i is the period of transaction x_i, TS_i is the set of all the tasks in x_i and $TS_i \subseteq \Gamma$. D_i is the deadline of x_i and $D_i \leq T_i$. Each task in TS_i has the same period T_i and deadline D_i.

If tasks do communicate with each other, they are said to be precedence constrained since a task is blocked until its direct predecessors activate it. Each transaction has one beginning task and or more terminative tasks. We denote *end(x)* the set of terminative tasks of transaction $x \in X$.

3 End-to-End Response Time and Release Jitter

The schedulability analysis for multiprocessor systems is based on the worst-case response time analysis of transactions. If the worst-case end-to-end response times of all the transactions in a system are less or equal to their deadlines, the system is then schedulable. To obtain the end-to-end response time of a transaction, we can calculate its individual task response times and sum them up [6].

We define *local* response time r_i for t_i which is measured from the local arrival time of t_i, and define *global* response time R_i of t_i in a transaction which is measured from the beginning of the complete transaction. The global worst-case response times of t_i, R^W_i can be expressed as:

$$R_i^W = R_{i-1}^W + r_i^W \qquad (1)$$

where r_i^W is the worst-case local response time of t_i, R^W_{i-1} is the worst-case global response time of its direct predecessor task and $R_0^W = 0$.

Obviously, the end-to-end response time of a transaction is its terminative tasks' global response time. Thus, the worst-case end-to-end response time R_x^W of transaction x can be expressed as:

$$R_x^W = \max_{j \in end(x)} (R_j^W) \tag{2}$$

In periodic multiprocessor systems, transactions composed of precedence-constrained tasks will begin periodically. However, tasks except the first in a transaction will suffer variation in release time since they will inherit variants in response time (jitter) from predecessor tasks. The presence of jitter can affect the response times of lower priority tasks [4]. Therefore, we need to accurately compute the release jitter for each task.

The release jitter of a task is defined as the maximum variation of its release time. Traditional analysis of distributed response time assumes that the release jitter of a task is the worst-case response time of the task which directly precedes it and the best-case response time of the predecessor task is assumed to be small and is ignored. However, this will lead to a pessimistic calculation of response times, since jitter may increase rapidly with each additional precedence step in a transaction. Given the global best-case response times of direct predecessors, the release jitter J_i of t_i can be more accurately computed through the following equation:

$$J_i = R_{i-1}^W - R_{i-1}^B \tag{3}$$

where R^B_{i-1} is the global best-case response times of direct predecessors of t_i. The calculation of best-case response time is not the focus in this paper. Readers can refer to [13] [14] and [15] for details.

Release jitters will be used in the calculation of response times of lower priority tasks on the same processor. In the following sections, we will describe the calculation of the worst-case response time of tasks in detail.

4 Schedulability Analysis

4.1 Traditional Worst-Case Analysis Technique

Audsley [4] and others [7] have studied the schedulability analysis used to derive worst-case response times of fixed priority periodic tasks preemptively scheduled. The worst-case response time of a task is computed assuming that it is released at the same instant as all higher priority tasks on the same processor -- this known as the "critical instant". If a task with higher priority suffers release jitter, the interval of its release time can be less than its period. This may lead to more preemption of tasks with lower priority. The local worst-case response time r_i^W of t_i can be expressed in the following iterative form:

$$r_i^{W*} = C_i^W + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i^W + J_j}{T_j} \right\rceil C_j^W \tag{4}$$

where $hp(i)$ is the set of tasks with higher priority than t_i, T_j and J_i are respectively the period and jitter of t_j, B_i is the longest time that t_i could be blocked by lower priority tasks, and is computed according to the concurrency control protocol, such as the priority ceiling protocol [2]. To calculate r_i^W, the newly computed response time, $r_i^{W}*$, replaces r_i^W on each iteration. The expression must be iterated until convergence ($r_i^{W*} = r_i^W$) or the task is deemed unschedulable ($r_i^W > D_i$). An initial response time, $r_i^W = C_i^W$, may be assumed.

4.2 Worst-Case Analysis for Tasks with Precedence Relations

When consider the precedence relations between tasks in the same processor, the above calculation may not provide the correct response time. Consider the task set on a processor p shown in Table 1. If the precedence constraints are not considered, r_2 and r_4 are 4 and 15 respectively based on Equation 4. If we exclude the preemption from t_2, r_4 is 8 as shown in Fig.1(a). However, this is not the worst-case phasing of t_4, since the precedence relations between tasks are not fully explored. The worst-case phasing of t_4 is shown in Fig.1(b), its local worst-case response time r_4 is 11. While the worst response time of t_2 will remain the same.

Table 1. The task set on processor *p*

Task	Transaction	Period	Direct predecessor	WCET	Priority
t_1	x_1	8	None	2	4
t_2	x_2	30	None	2	3
t_3	x_1	8	t_1	3	2
t_4	x_2	30	t_2	3	1

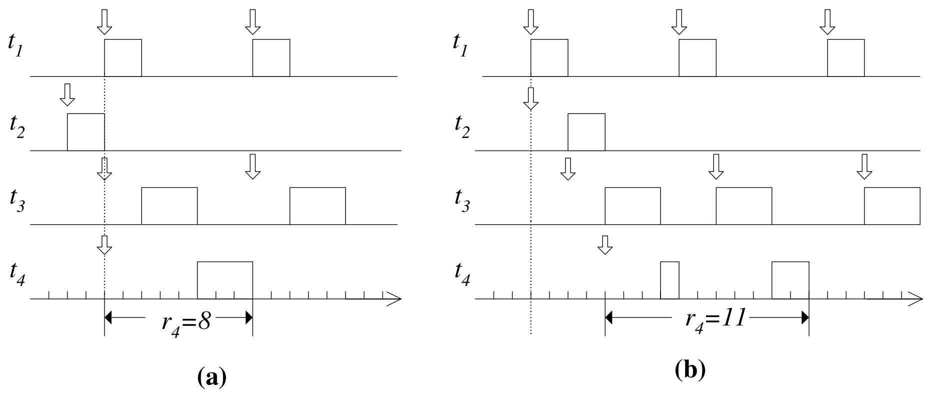


Fig. 1. The worst-case phasing of t_4

From this example we can see that the precedence relation between two tasks does not affect the execution of the predecessor, this is because the predecessor always has a higher priority than its successors on the same processor. The precedence relations between tasks on different processors have been considered in the calculation of global response time (Equation 1). In order to extend the response time analysis for precedence relations between tasks on the same processor, we consider the following two cases on a processor *p*:

Case 1: The calculated task t_i has no direct predecessor on *p*.

Case 2: The calculated task t_i has a direct predecessor on *p*.

Case 1: t_i has no direct predecessor on *p*. Let's consider the scheduling on *p*. When calculating t_i's response time, we do not consider the effects from its indirect predecessors on *p*, since they do not preempt the execution of t_i. Although these indirect predecessors may affect the release time of t_i, these effects are indirect and are transferred to t_i's direct predecessor, which is on other processor. The dependence on its direct predecessor has been taken into account in the calculation of global response time of t_i (see Equation 1). As a result, t_i's execution is only affected by the tasks with higher priority in other transactions. We denote the set of these tasks as *S*, so *S* can be expressed as $S = \{t_i \mid t_i \in hp(i) \wedge t_i \notin V(i)\}$, where *hp(i)* is the set of tasks on *p* which have higher priority than t_i and *V(i)* is the set of tasks in the transaction of t_i. If the tasks in *S* have no precedence relations among them (no two tasks belong to a common transaction), the precedence relations between tasks on *p* will not affect the execution of t_i. The worst-case phasing for t_i is the instant when it is released simultaneously with all the tasks in *S*.

Consider the case that two tasks in *S* have a precedence relation. Let the two tasks be t_j and t_k, and t_j is a predecessor of t_k. t_k may be released just when t_j terminate its execution (directly released by t_j) or released by successors of t_j (indirectly released by t_j). In the latter situation there is a delay between t_j's termination and the release of t_k. Obviously, the former situation will contribute to the worst-case response time of lower priority tasks (task t_i), since the interval between the executions of t_j and t_k is the least which leads to the tightest preemption of lower priority tasks. Thus we calculate t_i's response time in the former situation for the worst-case analysis.

Theorem 1: In the task set of a processor, task t_j is the direct predecessor of task t_k and t_k is released when t_j terminates its execution. If t_k's release time is brought forward to the instant when t_j is released, the execution of t_k is not changed.

Proof: According to our assumptions, t_j and t_k has a common period and t_j has a higher priority than t_k. Consider the situation illustrated in Fig.2 with $\pi_j > \pi_k > \pi_i$. The white arrow denotes the release instant of tasks when the precedence relations are considered. $a2$ is the instant when t_j is terminated and t_k is released. $a1$ is the instant when t_j is released and is denoted by the first black arrow. We can see that if t_k is released at $a1$, it will have no chance to execute before the termination of t_j in the same instance of their transaction. This is because that t_j and t_k are released simultaneously at $a1$ and t_j has a higher priority than t_k. That is to say the execution of t_k will not be different whether it is released at $a1$ or $a2$. Since t_j and t_k have the same period, this is true for all the following instance of t_k. ∎

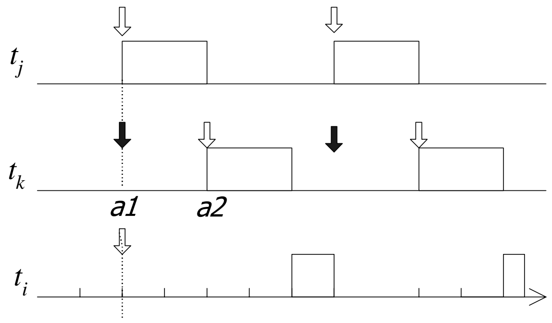


Fig. 2. Bringing forward t_k's release time to $a1$ dose not change t_k's execution

According to Theorem 1, the calculation of t_i's worst-case response time can be performed as if all the tasks in S were released at its critical instant, since the release time of successor tasks can be brought forward to the critical instant without changing their executions. So the precedence relations between tasks in S will not affect the worst-case phasing of t_i, which the instant when t_i is released simultaneously with all the tasks in S.

Thus, whether or no there are precedence relations between tasks in S, t_i's worst-case local response time r_i^W can be computed through the following iterative equation:

$$r_i^{W*} = C_i^W + \sum_{j \in hp(i) \wedge j \notin V(i)} \left\lceil \frac{r_i^W + J_j}{T_j} \right\rceil \cdot C_j^W \tag{5}$$

An initial response time, $r_i^W* = C_i^W$ may be assumed. The newly computed response time, r_i^W*, replaces r_i^W on each iteration. The expression must be iterated until convergence ($r_i^{W*} = r_i^W$) or the task is deemed unschedulable ($r_i^W > D_i$). Based on Equation 1, the worst-case global response time of t_i can be achieved.

Case 2: The calculated task t_i has a direct predecessor on p. To calculate the worst-case local response time of t_i, we need to consider the effects from its predecessors. We introduce the *consecutive sub-transaction* to compute the response time of t_i.

Definition 1: The *consecutive sub-transaction* of a task t is the sub-transaction consists of all the consecutive tasks ended up with t on the same processor.

In Fig.3, the consecutive sub-transaction of t_5 and t_6 are shown. We denote the consecutive sub-transaction of t_i as $ST(i)$. Obviously, t_i and $ST(i)$ have the same global response time. The worst-case global response time of t_i can be expressed as:

$$R_i^W = R_{fst(i)-1}^W + r_{ST(i)}^W \tag{6}$$

where $fst(i)$ is the first task in $ST(i)$, $r^W{}_{ST(i)}$ is the worst-case local response time of $ST(i)$ which is measured from the release time of the first task to the termination of the last task t_i in $ST(i)$.

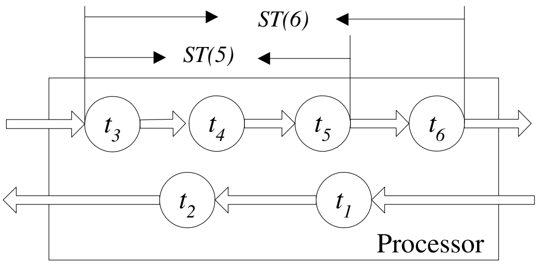


Fig. 3. Consecutive sub-transaction

To calculate $r^W{}_{ST(i)}$, we introduce the concept *canonical form* proposed by Harbour et al. [11]. A transaction or sub-transaction is said to be in canonical form if it consists of consecutive tasks that do not decrease in priority. Harbour et al. have proved that converting a precedence chain into a canonical form does not change its completion time. Therefore, we can calculate $r_{ST(i)}$ by transforming *ST(i)* to a canonical form. We denote the canonical form of *ST(i)* as *CST(i)*. The canonical transformation can be performed by applying the following algorithm:

Algorithm 1. Canonical transformation

```
$t_i$ is the last task in $ST(i)$;
while $t_i$ is not the first task of $ST(i)$
    If $\pi_{i-1} > \pi_i$ then $\pi_{i-1} = \pi_i$;
    $i = i-1$;
end
```

Since the consecutive tasks in a consecutive sub-transaction have descending priorities, after the canonical transformation, all the tasks in *CST*(*i*) have the same priority π_i. Due to the tasks in *CST(i)* also have the same period and each task is immediately released when its direct predecessor terminates its execution, *CST(i)* can be treated as a single task with priority π_i and period T_i when calculate its response time. According to the definition of *ST(i)*, this assumed task has no direct predecessor on *p*. This is just the case indicated by case 1. Therefore, the worst-case local response time $r_{CST(i)}$ (equal to $r_{ST(i)}$) can be expressed in the following iterative form:

$$r^{W*}_{CST(i)} = C^W_{CST(i)} + \sum_{j \in hp(i) \wedge j \notin V(i)} \left\lceil \frac{r^W_{CST(i)} + J_j}{T_j} \right\rceil \cdot C^W_j \tag{7}$$

where $C^W{}_{CST(i)}$ is the worst-case execution time of *CST(i)*, it can be achieved by summing up the worst-case execution time of all the task in *CST(i)*. This equation is deduced from Equation 6 by treating *CST(i)* as one task without direct predecessor. Please note that the preemptions from the predecessors of *ST(i)* are excluded from the interference time (the item on the right of the equation) since *ST(i)* are always released after their terminations and they have no chance to preempt it.

To calculate $r^W_{CST(i)}$, the newly computed response time, $r^{W}*_{CST(i)}$, replaces $r^W_{CST(i)}$ on each iteration. The expression must be iterated until convergence ($r^{W}*_{CST(i)} = r^W_{CST(i)}$) or the task is deemed unschedulable ($r^W_{CST(i)} > D_i$). An initial response time, $r^W_{CST(i)} = C^W_{CST(i)}$, may be assumed. Since transforming *ST(i)* to *CST(i)* does not change its completion time, $r^W_{ST(i)}$ is equal to $r^W_{CST(i)}$. Based on Equation 6, the worst-case global response time of t_i can be achieved.

4.3 An Integrated Schedulability Analysis Algorithm

We extend the algorithm proposed by Henderson et al. [13] to test the schedulability of multiprocessor systems. The analysis is based on the computation of worst-case response time for end-to-end transactions, which essentially involves the two stages:

1. Compute local response time for task or consecutive sub-transactions on all processors
2. Compute global response times and jitter for tasks on all processors

These stages are repeated until convergence is achieved. The second stage requires the traversal of precedence graphs to sum the delays along each end-to-end transaction. Because not all information is available when computing response times on each processor, the process has to proceed by iteration. The bounded response time of each transaction and the schedulability analysis of the system can be achieved using the following algorithm:

Algorithm 2. SAPR (Schedulability analysis for task with precedence relations)

```
begin
  Initialize global responses, $R_0^W = 0$;
  Initialize jitter, $J_0 = 0$;
  do
    for each $p_k \in P$ do
      for each $t_i \in \Gamma_{pk}$ do
        if $t_i$ has no direct predecessor on $p_k$ then
          Compute local task response time, $r_i^W$ (Eq. 5)
        else
          Compute local response time of $ST(i)$, $r^W_{ST(i)}$; (Eq. 7)
      end for
    end for
    for each transaction $x$ do
      for each task $t_i$ in $x$ do
        Compute global response time, $R_i^W$; (Algorithm 3)
        Compute jitter, $J_i$; (Eq. 2)
      end for
    end for
```

```
        Test for convergence;
      until convergence or not schedulable
  end
```

Task Global response times can be calculated through the following nested algorithm:

```
Algorithm 3. Task global response time calculation
input: $x$, $p$, $t_i$, $r_i^W$, $r^W_{ST(i)}$;
output: $R_i^W$;
begin
    if $t_i$ is the beginning time of $x$ then
        $R_i^W = r_i^W$;
    else
        if $t_i$ has no direct predecessor on $p$ then
            compute $R_{i-1}^W$; (Algorithm 3)
            compute $R_i^W$ and $R_i^B$; (Eq. 1)
        else
            compute $R_{fst(i)}^W$; (Algorithm 3)
            compute $R_i^W$; (Eq. 6)
        end if
    end if
end
```

In the SAPR algorithm (Algorithm 2), the convergence test is satisfied when all response times remain the same from one iteration to the next. For many small-medium sized systems, between 2 and 6 iterations appear sufficient to obtain a converged solution.

5 Simulations

In order to evaluate the performance of the proposed method, simulations were conducted on a Pentium 4 2.4GHz with different task sets whose execution times, periods and priorities were generated randomly. We have compared the SAPR algorithm with the dynamic offset algorithm proposed by Palencia and Harbour [10] through simulations and found that both the algorithms can achieve the same result for a given task set when BCET of tasks is considered to be zero. However, the dynamic offset algorithm consumes much more time than SAPR to obtain the results for medium or large size systems. Fig.4 shows the time cost of the dynamic offset algorithm to analyze a system with 5 transactions in 2 processors. It can be seen that the time cost of the dynamic offset algorithm grows quickly with the increase of task amount and the processor utilization. Even for a task set with 40 tasks and a normal

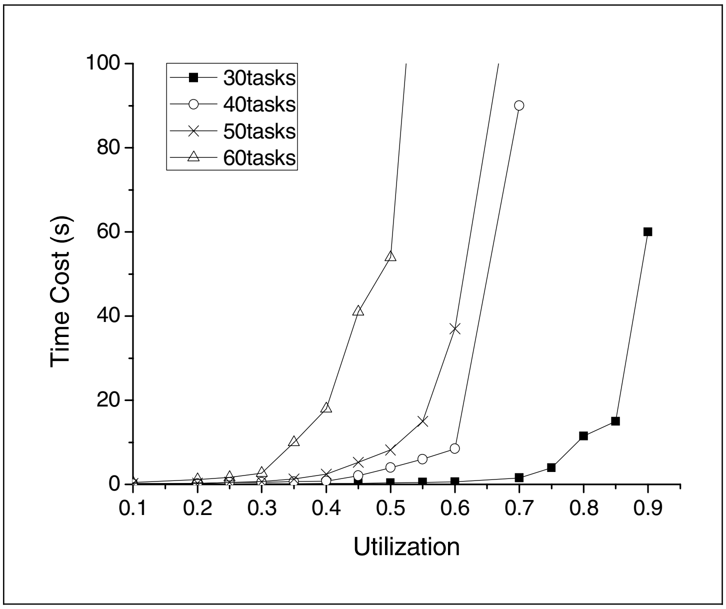


Fig. 4. Time cost of dynamic offset algorithm for 5 transactions on 2 processors

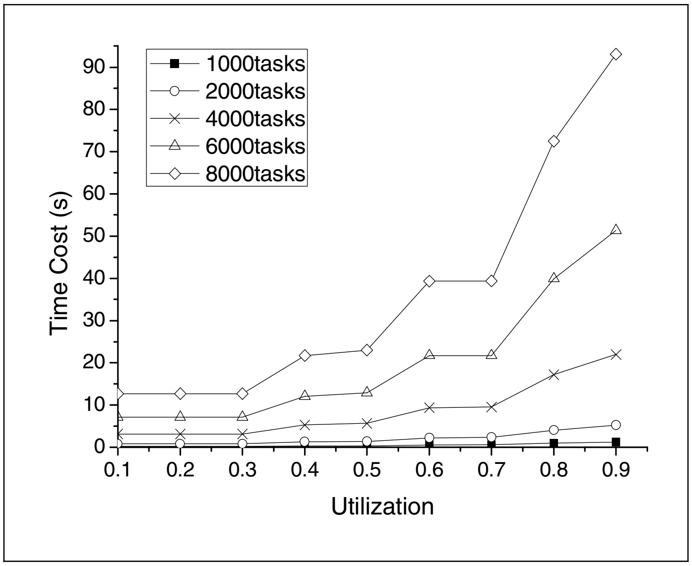


Fig. 5. Time cost of SAPR algorithm for 5 transactions on 2 processors

utilization level of 0.7, it will take more than one minute to achieve the result. This performance cannot be tolerated in practice for many larger size systems. The SAPR algorithm has a higher performance than the dynamic offset algorithm. Fig.5 shows its time cost to analyze the systems with the same amount of transactions and processors but much more tasks. For a task set with 1000 tasks, the analysis can be completed in less than 2 seconds even under very high processor utilization levels close to 1. It can also be noted that the time cost of the SAPR algorithm grows slowly with the increase of amount of tasks and the processor utilization. These features make it quite applicable to large size systems.

6 Conclusion

In this paper we have addressed the problem of schedulability analysis in hard real-time distributed systems. In safety-critical control systems, periodic control actions are required with precise timing and accurate performance data is required to assure the behavior of these systems. To achieve an accurate analysis, the existing time analysis techniques are extended to exploring the precedence relations between tasks in the calculation of the worst-case response times. An integrated algorithm is also proposed to test the schedulability and compute the response time bounds of a system. A simulation study has shown that this algorithm can provide an accurate analysis and offer good performance for systems with wide range of CPU utilizations and task set size, and therefore is suitable for analysis of complex systems. Although the costs of message transmission is not taken into account, the present analysis technique can also be used for communications on networks by modeling each network as if it were a processor, and each message as if it were a task [12].

Our further work includes the optimization of scheduling for distributed systems based on the end-to-end response time analysis. This will help system designers to improve the performance and reliability of a complex system at an early design stage.

Acknowledgements

This research was supported by 863 National High Technology Program under Grant No. 2004AA1Z2180, No. 2003AA1Z2080 and No. 2003AA1Z2140.

References

1. C. L. Liu, and J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. 20(1), Journal of the ACM (1973) 46-6
2. L. Sha, R. Rajkumar, and S. Sathaye: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers, Vol.39, (1990) 1175-1185
3. N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings: Hard Real-Time Scheduling: The Deadline Monotonic Approach. Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, (1991) 15-17
4. N. C. Audsley, A. Burns, K. Tindell, M. Richardson, A. Wellings: Applying New Scheduling Theory To Static Priority Pre-emptive Scheduling. Software Engineering Journal, 8(5), (1993) 284-292
5. J. P. Lehoczky: Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines. Proceedings 11th IEEE Real-Time Systems Symposium, (1990) 201-209

6. M. Klein, T. Ralya, B. Pollack, R. Obenza andM. González Harbour: A Practitioners Handbook for Real-Time Systems Analysis. Kluwer Academic Publishers, (1993)
7. K. Tindell and J. Clark: Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. Microprocessing & Microprogramming, 50(2-3), (1994) 117-134
8. J. C. Palencia and M. Gonzalez Harbour: Schedulability analysis for tasks with static and dynamic offsets. In Proc. of IEEE Real-time Systems Symposium, (1998) 26-37
9. J.C. Palencia Gutiérrez, J.J. Gutiérrez García and M. González Harbour: Best-Case Analysis for Improving the Worst-Case Schedulability Test for Distributed Hard Real-Time Systems. Proceeding Of tenth Euromicro Workshop on Real-Time Systems, (1998) 35-44
10. J. C. Palencia and M. Gonzalez Harbour: Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In Proc. of IEEE Real-time Systems Symposium, (1999) 328-339
11. M.G. Harbour, M.H. Klein, and J.P. Lehoczky: Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In Proc. of Real-Time Systems Symposium, San Antonio, (1991) 116-128
12. Tindell, K., Burns, A., and Wellings, A. J.: Analysis of hard real-time communications. Real-Time Systems, Vol.9, (1995) 147–171
13. W. Henderson, D. Kendall and A. Robson: Improving the Accuracy of Scheduling Analysis Applied to Distributed Systems. Journal of Real-Time Systems, Kluwer, 20(1), (2001) 5-25
14. O. Redell and M. Sanfridson: Exact Best-Case Response Time Analysis of Fixed Priority Scheduled Tasks. In the 14th Euromicro Conference on Real-Time Systems, Vienna (Austria), (2002) 165-172
15. P. E. Hladik and A. M. Deplanche: Best-Case Response Time Analysis for Precedence Relations in Hard Real-Time Systems. Real-Time Systems Symposium Work-in-Progress Session, (2003), Available: http://www.cs.virginia.edu/~zaher/rtss-wip/22.pdf

Safety Interfaces for Component-Based Systems

Jonas Elmqvist[1], Simin Nadjm-Tehrani[1], and Marius Minea[2]

[1] Department of Computer and Information Science, Linköping University
{jonel, simin}@ida.liu.se
[2] "Politehnica" University of Timişoara and Institute e-Austria Timişoara
marius@cs.utt.ro

Abstract. This paper addresses the problems appearing in component-based development of safety-critical systems. We aim at efficient reasoning about safety at system level while adding or replacing components. For safety-related reasoning it does not suffice to consider functioning components in their "intended" environments but also the behaviour of components in presence of single or multiple faults.

Our contribution is a formal component model that includes the notion of a safety interface. It describes how the component behaves with respect to violation of a given system-level property in presence of faults in its environment. We also present an algorithm for deriving safety interfaces given a particular safety property and fault modes for the component. Moreover, we present compositional proof rules that can be applied to reason about the fault tolerance of the composed system by analyzing the safety interfaces of the components. Finally, we evaluate the above technique in a real aerospace application.

1 Introduction

Component-based software development [30,9] uses various models and methods to capture different attributes of a system, or emphasise phases of the development cycle [4,28,8,27,10]. This paper addresses efficient assurance of dependability in a system built from components and with several upgrades in its life cycle, an aspect not widely studied so far in the components literature [11].

Modifying a component or replacing it with another is an especially costly process for safety-critical systems, as much of the analysis and review of the safety arguments at the certification stage has to be repeated for every significant change to the system. We believe that tool support in this sector needs to make component changes cost-efficient by addressing safety-specific issues, e.g. resilience of the system with respect to single and multiple faults as new components are plugged in. The model we propose covers digital components, with a built-in declaration of their behaviour under faults in assumed environments. This component model captures the logic of the design at a high abstraction level, and could be applied to software or (reconfigurable) hardware designs.

Traditional risk assessment techniques such as Fault-tree analysis (FTA) and Failure modes and effects analysis (FMEA) [16] deal with the effect of independent faults. Although assessing fault tolerance at system level is an important part of safety analysis, rigorous methods are only in their infancy when it comes to systems with significant digital components [15,13]. Our goal is to provide a formal means to support the system integrator. When acquiring a new component for inclusion into a system, the integrator is informed whether the component can potentially threaten the system-level safety (in the same spirit as FMEA). The integrator is also supported in analysis of fault tolerance at system level, the result of which will indicate all single or multiple component-level faults that will necessarily lead to violation of safety (in the same vein as FTA). Unlike functional correctness analysis, here the goal is to focus on risks associated with external faults, not to eliminate design faults.

The contributions of this paper are as follows. We present a component model that includes *safety interfaces*. These describe how a component behaves with respect to a given system-level safety property in presence of (a defined set of) faults in its environment. We show how to perform a system-level safety analysis by using the safety interfaces of components. This goal is supported in two ways. First, we provide an algorithm that derives the safety interface of a component given a particular safety property and set of fault modes. The interface includes the single and multiple faults that this component is resilient to, as well as environment restrictions that can contain the faults. This analysis is intended to be performed by the developer of the component. Second, we support the system integrator to reason compositionally about safety in presence of single and multiple faults at system level by referring to the safety interfaces. Once the relevant fault-failure chains are rigorously identified, they can be handled using standard assessment routines, fault forecasting and containment techniques [20].

1.1 Related Work

To our knowledge, there is no previous formal work on safety interfaces for components.

Current engineering practice includes two parallel activities for safety-related studies (hazard analysis, FTA and FMEA) and functional design and analysis. Recent research efforts have tried to combine these separate tracks by augmenting the system design model with specific fault modes. Åkerlund et al. [2] have to our knowledge the first attempt to integrate the separate activities of design and safety analysis and support them by common formal models. Hammarberg and Nadjm-Tehrani extend this work to models at a higher level of abstraction and characterise patterns for safety analysis of digital modules [15]. That work, however, does not build on a notion of encapsulation as in components. It verifies the entire composed system in Esterel using a SAT model checker and iteratively analyses all fault modes at system level. The ESACS project [7] applies a similar approach to Statechart models using a BDD-based verification engine.

Papadopolous et. al. [24] extend a functional model of a design with Interface-Focused FMEA. The approach follows a tabular (spread sheet) editor layout. The formalised syntax of the failure classes allows an automatic synthesis of a fault tree and incorporation of knowledge about the architectural support for mitigation and containment of faults. However, it suffers from combinatorial explosion in large fault trees and lacks formal verification support.

Rauzy models the system in a version of mode automata and the failure of each component by an event that takes the system into a failure mode [26]. The formal model is compiled into Boolean equations and partial order techniques are suggested for reducing the combinatorial explosion. However, it has not been applied to component-based development or compositional reasoning.

Strunk and Knight define the system and its reconfiguration elements explicitly using RTL (temporal logic) notation and provide guidelines for reconfiguration assurance. Reconfiguration is mainly used here when the system is adapting to lower service levels that may in particular be due to failure scenarios [29].

Jürjens defines an extension of the UML syntax in which stereotypes, tags, and values can be used to capture failure modes of components in a system (corruption, delay, loss) [19]. The merit of the model is to narrow the gap between a system realised as a set of functions and a system realised as a set of components.

Li et al. [21] define feature-oriented interfaces for modules that encapsulate crosscutting system properties. The focus of this work is feature interaction including features that introduce a new vocabulary.

A recent approach for formal treatment of crosscutting concerns in reconfigurable components is given by Tesanovic et al. [31] where extended timed automata are used to capture models of components with an interface for characterising the essential traces for supporting a given *timing property*.

Assume-guarantee-style compositional reasoning has a long history originating with the work by Misra and Chandy [23] and Jones [18] in the context of concurrent systems. It has been applied to deductive reasoning about specifications [1] as well as model checking for various automata formalisms. Here, the notion of refinement is usually trace inclusion, but can also be simulation [17]. Our rules are derived from those of Alur and Henzinger for reactive modules [3].

2 Components and Fault Models

A component is an independent entity that communicates through well-defined interfaces. In most component models, the interfaces are only functional, defining input and output ports at a syntactic level. For efficient safety analysis at system level, these simple interfaces are insufficient. More behaviour information must be provided to make interfaces usable for analysis of failures in presence of faults.

We propose a formal component model with two elements: its functional behaviour and a *safety interface*, which describes the behaviour in presence of faults in the environment. This safety interface can then be used to perform safety analysis at system level, such as analysis for fault tolerance. We next present the basic definitions, the fault modes and the employed formalism.

2.1 Modules and Basic Definitions

Our general formalism for modules is based on the notion of *reactive modules* [3], of which we give only a brief overview. We present a special class of reactive modules with synchronous composition, finite variable domains and non-blocking transitions that we call *synchronous modules* (by default, simply modules).

A module is defined by its *input*, *output* and *private variables* and the rules for updating and initializing them. Variables are updated in a sequence of rounds, each once per round. To model synchrony, each round is divided into subrounds, and the system and the environment take turns in executing and updating variables. Events, such as a *tick*, can be modelled by toggling boolean variables.

Definition 1 (Module). *A synchronous module M is a tuple (V, Q_0, δ) where*

- *$V = (V_i, V_o, V_p)$ is a set of typed variables, partitioned into sets of input variables V_i, output variables V_o and private variables V_p. The controlled variables are $V_{ctrl} = V_o \cup V_p$ and the observable variables are $V_{obs} = V_i \cup V_o$;*
- *A* state *over V is a function mapping variables to their values. The set of controlled states over V_{ctrl} is denoted Q_{ctrl} and the set of input states over V_i as Q_i. The set of states for M is $Q_M = Q_{ctrl} \times Q_i$;*
- *$Q_0 \subseteq Q_{ctrl}$ is the set of initial states;*
- *$\delta \subseteq Q_{ctrl} \times Q_i \times Q_{ctrl}$ is the* transition relation.

The successor of a state is obtained at each round by updating the controlled variables of the module. The execution of a module produces a state sequence $\bar{q} = q_0 \dots q_n$. A trace $\bar{\sigma}$ is the corresponding sequence of observations on $\bar{q}$, with $\bar{\sigma} = q_0[V_{obs}] \dots q_n[V_{obs}]$, where $q[V']$ is the projection of q onto a set of variables $V' \in V$. The trace language of M, denoted $\mathcal{L}_M$, is the set of traces of M.

A property φ on a set of variables V is defined as a set of traces on V. A module M satisfies a property φ, written $M \models \varphi$, if all traces of M belong to φ. This work focuses on *safety properties* [22,14] as opposed to liveness properties.

Composing two modules into a single module creates a new module whose behaviour captures the interaction between the component modules.

Definition 2 (Parallel composition). *Let $M = (V^M, Q_0^M, \delta^M)$ and $N = (V^N, Q_0^N, \delta^N)$ be two modules with $V_{ctrl}^M \cap V_{ctrl}^N = \emptyset$. The parallel composition of M and N, denoted by $M \parallel N$, is defined as*

- *$V_p = V_p^M \cup V_p^N$*
- *$V_o = V_o^M \cup V_o^N$*
- *$V_i = (V_i^M \cup V_i^N) \setminus V_o$*
- *$Q_0 = Q_0^M \times Q_0^N$*
- *$\delta \subseteq Q_{ctrl} \times Q_i \times Q_{ctrl}$ where $(q, i, q') \in \delta$ if $(q[V_{ctrl}^M], (i \cup q)[V_i^M], q'[V_{ctrl}^M]) \in \delta^M$ and $(q[V_{ctrl}^N], (i \cup q)[V_i^N], q'[V_{ctrl}^N]) \in \delta^N$.*

We extend Definition 2 to a pair of modules with shared outputs, provided the resulting transition relation δ is *nonblocking*, i.e., has a next state for any combination of current state and inputs. In this case, we call the two modules *compatible* and distinguish *nonblocking composition* by denoting it $\widehat{\parallel}$.

We relate modules via trace semantics: a module M refines a module N if N has more behaviours than M, i.e., all possible traces of M are also traces of N.

Definition 3 (Refinement). *Let $M = (V^M, Q_0^M, \delta^M)$ and $N = (V^N, Q_0^N, \delta^N)$ be two synchronous modules. M refines N, written $M \leq N$, if (1) $V_o^N \subseteq V_o^M$, (2) $V_{obs}^N \subseteq V_{obs}^M$ and (3) $\{\bar{\sigma}[V_{obs}^N] : \bar{\sigma} \in \mathcal{L}_M\} \subseteq \mathcal{L}_N$.*

2.2 Fault Mode Models

To analyse the behaviour of a component in presence of faults in its environment it is important to identify all possible ways that the environment can fail. Low-level fault modes are generally application and platform dependent, however faults can be classified into high-level categories. Bondavalli and Simoncini classify faults into *omission faults*, *value faults* and *timing faults* [6]. We adopt a classification in which faults fall into the following categories [12,25,24]:

Omission failure i.e., absence of a signal when the signal was expected.
Commission failure i.e., unexpected emission of a signal.
Value failure i.e., failure in the value domain such as signal out of range or a signal stuck at some value etc.
Timing failure i.e., failure in the time domain such as late or early delivery.

Timing properties have been addressed in other work, for example interfaces to capture timing properties in the absence of faults are given by Tesanovic et al. in timed automata [31] and could be further extended to cover resilience to timing failures. In this work we focus on untimed models and value failures.

We model faults in the environment as delivery of faulty input to the component and call each such faulty input a *fault mode* for the component. A value failure is modelled by modifying the input signals that in turn might affect private variables. A commission failure is modelled by unforced emission of signals to the component. The input fault of one component thereby captures the output fault of a component connecting to it, with the exception of "edge" components that need to be treated separately, e.g. in accordance to earlier methods [15].

Definition 4 (Input Fault Mode). *An input fault mode F_j of a module M is a module with an input variable $v_j^f \notin V^M$, an output variable $v_j \in V_i^M$, both of the same type D, and an unconstrained transition relation $\delta = D \times D \times D$.*

A fault mode F_j on the input v_j from environment E to module M can be viewed as replacing the original output v_j of E with the input v_j^f of F_j, which produces the faulty output v_j to M. We model this formally as a composition of F_j and E, which has the same variables as E and can then be composed with M. Free inputs to M are viewed as unconstrained outputs of E.

Definition 5 (Composition with Fault). *Let E be a module with $v_j \in V_o^E$ and F_j a fault mode with output v_j and input v_j^f. Denote $F_j \circ E = F_j \parallel E[v_j/v_j^f]$ where $E[v_j/v_j^f]$ is the module E with the variable substitution v_j^f for v_j.*

Our fault modes are unrestricted and can affect their output in an arbitrary way. Other types of fault modes can be modelled by appropriate logic in their transition relation. We can naturally extend this definition to multiple faults.

2.3 Components and Safety Interfaces

Given a module, we wish to characterize its fault tolerance in an environment that represents the remainder of the system together with any external constraints. Whereas a module represents an implementation, we wish to define an interface that provides all information about the component that the system integrator needs. Traditionally, these interfaces do not contain information about safety of the component. In this paper we propose a safety interface that captures the behaviour of the component in presence of faults in the environment.

Definition 6 (Safety Interface). *Given a module M, a system-level safety property φ, and a set of fault modes F for M, a safety interface SI^{φ} for M is a tuple $\langle E^{\varphi}, \mathsf{single}, \mathsf{double}\rangle$ where*

- *E^{φ} is an environment in which $M \parallel E^{\varphi} \models \varphi$.*
- $\mathsf{single} = \langle F^s, E^s\rangle$ *where $F^s \subseteq \mathcal{P}(F)$ is the* single fault resilience set *and E^s is a module composable with M, such that $\forall F_k \in F^s, M \parallel (F_k \circ E^s) \models \varphi$*
- $\mathsf{double} = \{\langle F^d_1, E^d_1\rangle, \ldots, \langle F^d_n, E^d_n\rangle\}$ *with $F^d_k = \langle F^1_k, F^2_k\rangle$, $F^1_k, F^2_k \in F$, $F^1_k \neq F^2_k$ such that $M \parallel ((F^1_k \parallel F^2_k) \circ E^d_k) \models \varphi$*

The safety interface makes explicit which single and double faults the component can tolerate, and the corresponding environments capture the assumptions that M requires for resilience to these faults. For single faults, we specify *one* environment assumption E^s under which the component is resilient to *any* fault from a given set of interest. For double faults, we are more fine-grained and specify for each fault pair of interest an environment in which the module is resilient to their joint occurrence. Multiple faults could be handled similarly. The safety interface need not cover all possible faults (and in fact could be empty): the provider of a component only specifies what is explicitly known about it.

Definition 7 (Component). *Let φ be a system-level safety property, M a module and SI^{φ} a safety interface for M. A component C is the tuple $\langle M, SI^{\varphi}\rangle$.*

We wish to deliver a component with precomputed information about the set of tolerated fault modes. To check safe use of the component one verifies that the actual environment satisfies the component assumptions which guarantee safety under faults.

3 Deriving Safety Interfaces

In this section we provide guidelines for how a component developer creates the safety interface. The developer needs to characterise environments in which a module functions correctly in the presence of a given set of faults. We first derive such an environment (in fact, the most general one) in the ideal case without faults. Next, we use the obtained environment abstraction to determine more restrictive environments under which the module is resilient, first to a chosen set of single faults and then for the occurrence of fault pairs.

3.1 Generating a Constraining Environment

If M is a module such that $M \not\models \neg\varphi$, the weakest (least restrictive) environment E^{φ}_w in order to satisfy φ can be generated as shown in Figure 1. The algorithm

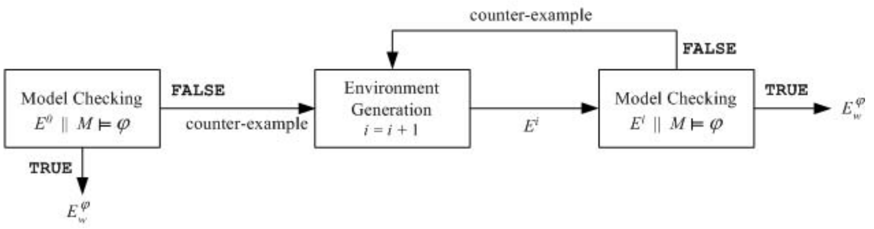


Fig. 1. The abstraction algorithm

uses a model checker to check whether the module M in parallel with an environment E satisfies the safety property φ; i.e. $M \parallel E \models \varphi$.

Initially, the algorithm starts out with an empty constraint E^0 on the environment and at each iteration i, the algorithm strengthens the constraints E^i by analysing the counter-example generated by the model checker and removing the forbidden states. This corresponds to removing behaviour from (or strengthening) the environment. In the next iteration, the environment E^{i+1} should at least not exhibit the behaviour reflected by the counter-example at iteration i. The algorithm stops at a fixpoint when $E^{i+1} = E^i = E_w^\varphi$.

Proposition 1. *The environment E_w^φ generated by the algorithm is the* least restrictive environment *in which M satisifies the property φ. That is, for any environment E, $M \parallel E \models \varphi$ iff $E \leq E_w^\varphi$.*

The proof can be done by adapting the reasoning by Halbwachs et. al. [14], that synthesise a necessary and sufficient environment for an I/O machine M.

3.2 Identification of Fault Behaviours

Let M be a module that satisfies a safety property φ when placed in an environment E, assuming the ideal case without faults: $M \parallel E \models \varphi$. Let F_j be a fault mode on variable v_j which is an input to M from E. Denote by $E' = \forall v_j \, E$ the module with $V^{E'} = V^E \setminus v_j$, $Q_0^{E'} = \forall v_j \, Q_0^E$ and $\delta^{E'} = \forall v_j \, \forall v'_j \, \delta^E$.

Proposition 2. *If $M \parallel E \models \varphi$ and $\forall v_i \, E$ exists, then $M \parallel (F_i \circ \forall v_i \; E) \models \varphi$, i.e., M is resilient to fault F_i in the environment $\forall v_i \, E$.*

By definition of $\forall v_i \, E$, any state can be extended to a state of E with an arbitrary value of v_i. Thus, $F_i \circ \forall v_i \, E \leq E$ and the result follows by composing with M. In particular, if E_w^φ is the least restrictive environment for M and φ, then $\forall v_i \, E_w^\varphi$ is the least restrictive environment in which module M is resilient to fault F_i.

This result gives an environment in which a module is resilient to a single fault. For the safety interface, we need an environment E^s which makes the module resilient to *any one* fault from the *single fault resilient set.* The desired environment must be at least as restrictive as the environment required for resilience of each of the individual fault modes. This is ensured by their parallel composition.

Proposition 3. *If $M \parallel (F_i \circ E_i) \models \varphi$ and $M \parallel (F_j \circ E_j) \models \varphi$, and E_i, E_j are compatible, then $M \parallel (E_i \mathbin{\widehat{\parallel}} E_j)$ is resilient to any fault F_i or F_j individually.*

This follows since any trace of $E_i \mathbin{\widehat{\parallel}} E_j$ under fault F_i or F_j is either a trace of $F_i \circ E_i$ or of $F_j \circ E_j$. In particular, we can take $E_i = \forall v_i\, E_w^\varphi$ with E_w^φ the least restrictive environment as determined in the previous section. Successive application to each fault in the selected set yields $E^s = \forall v_i\, E_w^\varphi \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} \forall v_n\, E_w^\varphi$ as the desired environment for the single element of the safety interface.

For resilience to double faults F_i and F_j, the environment must be restricted, analogously to Proposition 2, to behaviours allowed for *all* values of v_i and v_j:

Proposition 4. *If $M \parallel E \models \varphi$, and F_i, F_j are faults such that $\forall v_i \forall v_j\, E$ exists, then $M \parallel ((F_i \parallel F_j) \circ \forall v_i \forall v_j\, E) \models \varphi$. That is, M is resilient to simultaneous faults F_i and F_j in the environment $\forall v_i \forall v_j\, E$.*

Thus, if $\forall v_i \forall v_j\, E$ is nonempty, the pair $\langle\langle F_i, F_j\rangle, \forall v_i \forall v_j\, E\rangle$ can be included in the double fault resilience portion of the safety interface. Moreover, if E_w^φ is the least restrictive environment for M and φ, then $\forall v_i \forall v_j\, E_w^\varphi$ is the least restrictive environment in which M is simultaneously resilient to F_i and F_j.

Example: Suppose module M guarantees the safety property φ if the environment E ensures that of the two boolean inputs v_1 and v_2, at least one is set to 1: $v_1 \vee v_2 = 1$. Then, the faulty environments become $E_1 = F_1 \circ E \equiv \forall v_1\, E \equiv \forall v_1 \,.\, v_1 \vee v_2 = 1 \equiv v_2 = 1$ and $E_2 = F_2 \circ E \equiv v_1 = 1$. The environment which is resilient to either fault is $E_1 \mathbin{\widehat{\parallel}} E_2 \equiv v_1 = 1 \wedge v_2 = 1$. There is no environment under which the module is resilient to a double fault.

4 Component-Based Analysis of Fault Tolerance

We next describe the methodology of applying the above component model in system safety analysis. Unlike component models that capture functional contracts as interfaces, and then apply assume-guarantee reasoning for ensuring that the system behaves functionally correct when built from given components, our model does not aim to prove the satisfaction of a property. Rather, the purpose of our analysis is to focus on sensitive faults. In other words, both resilience and non-resilience information are interesting in the follow-up decisions. If the system safety is indeed threatened by a single fault, then the systems engineer may or may not be required to remove the risk of that fault by additional actions. This typically implies further qualitative analysis of the risk for the fault and its consequence, and is outside the scope of this paper. Combinations of multiple faults typically bear a lower risk probability but as important to quantify and analyse. If the safety of the system is not sensitive to a fault (pair), then the engineer can confidently concentrate on other combinations of potential faults that are a threat. In general, it is likely that none of these faults appear in actual operation, and the whole study is only hypothetical in order to provide arguments in preparing the safety case for certification purposes.

With this introduction, we will now proceed to explain the steps needed to ascertain the sensitivity of the system to single (respectively multiple) faults.

4.1 General Setup

Consider a system safety property φ and a component with safety interface SI^{φ}. As delivered by the component provider, SI^{φ} specifies an environment in which the component is safe, assuming no faults; another environment in which the component is resilient to a set of single faults, and a safe environment for each considered pair of simultaneous faults. Consider proving $M_1 \parallel M_2 \parallel ... \parallel M_n \models \varphi$ in the presence of a fault F_j in M_1. If a safety interface SI^{φ} of M_1 is known, with $\mathsf{single} = \langle F_1^s, E_1^s \rangle$, and $F_j \in F_1^s$, it suffices to show that $M_2 \parallel \ldots \parallel M_n \leq E_1^s$.

However, composing all modules is against the idea of modular verification. This can be overcome using circular assume-guarantee rules [3], for which we first derive an n-module version. The rule requires that every module in its environment (an abstraction of the other modules) refines each other environment. We can then infer that the system refines the composition of the environments without paying the price of an expensive overall composition, and without having to redo the entire analysis each time a component changes.

Lemma 1. *Let M_j and E_j, $1 \leq j \leq n$ be modules and environments such that the compositions $I = M_1 \parallel \ldots \parallel M_n$ and $E = E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$ exist and $V_j^E \subseteq V_{obs}^I$. Then, if $\forall i \forall j\ M_j \parallel E_j \leq E_k$ we have $M_1 \parallel \ldots \parallel M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$.*

In concise rule form: $$\frac{\forall j \forall k\ M_j \parallel E_j \leq E_k}{M_1 \parallel \ldots \parallel M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n}$$

The proof follows that of Proposition 5 in [3], showing inductively that every step of the implementation I can be extended to a step of the specification E. Requiring nonblocking composition guarantees soundness despite circularity.

To reason compositionally about safety, we add n premises stating that each module in its given environment satisfies the safety property: $\forall i\ M_i \parallel E_i \models \varphi$. Together with the premises above, we can then prove safety for the composition:

Proposition 5. *If M_j and E_j, $1 \leq j \leq n$ satisfy the conditions of Lemma 1 and in addition $M_j \parallel E_j \models \varphi$ for $1 \leq j \leq n$ then we have $M_1 \parallel M_2 \parallel \ldots \parallel M_n \models \varphi$.*

In concise form: $$\frac{\forall j\ M_j \parallel E_j \models \varphi \qquad \forall j \forall k\ M_j \parallel E_j \leq E_k}{M_1 \parallel M_2 \parallel \ldots \parallel M_n \models \varphi}$$

Proof. Composing $M_j \parallel E_j \models \varphi$ for $j = 1...n$ we get $I \mathbin{\widehat{\parallel}} E \models \varphi$. By Lemma 1, $M_1 \parallel \ldots \parallel M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$, or $I \leq E$. Thus $I \mathbin{\widehat{\parallel}} I \models \varphi$ or $I \models \varphi$.

This rule provides a generic assume-guarantee framework, independent of faults. We need to discharge n^2 premises to prove the global property φ, but each of those involves only one module, and at most two environment abstractions, assumed to be much smaller than the global composition. To use the rule, we need to find appropriate environments E_i, and to apply it to system safety, the environments must make the premises hold even with the analyzed fault(s) occurring. We derive these environments from the component safety interfaces.

4.2 Single Faults

We assume that single faults affect only one component. Using the environments E^φ and E^s, we check safety compositionally showing the premises of Prop. 5:

- a module in a faulty environment still provides an environment that guarantees the safety of each other module in absence of another fault
- a module in a non-faulty environment provides for each other module the environment of the SI which makes it resilient to single faults.

Thus, we need to show premises (a) $M_j \parallel F \circ E_j^s \leq E_k^\varphi$ and (b) $M_k \parallel E_k^\varphi \leq E_j^s$, ranging over modules M_j with potential faults F, and non-faulty modules M_k. If the interface provides the weakest environment E_w^φ, the fault-specific premises (a) can be jointly replaced by $M_j \parallel E_j^\varphi \leq E_k^\varphi$, with fewer obligations to discharge.

4.3 Multiple Faults

Next we study whether two faults F_a and F_b appearing in *different* components can, together, violate system safety. In Proposition 5 we use different environments for each component, depending on how they are affected by faults:

- for a module M_i affected by both faults, we check whether the double part of the safety interface contains a tuple $\langle\langle F_a, F_b\rangle, E_i^{ab}\rangle$, and use environment E_i^{ab}.
- for a module M_j with only one fault F_a, we use environment $E^a = \forall v_a E_j^\varphi$.
- for a module M_k not affected by faults, we use the environment E_k^φ.

Here, we have more fault-specific premises, but since environments for double faults are more restrictive, some premises can subsume or be subsumed by those for single faults. Thus, $M_i \parallel E_i^{ab} \leq E_k^\varphi$ follows from $M_i \parallel E_i^a \leq E_k^\varphi$, and checking $M_k \parallel E_k^\varphi \leq E_i^{ab}$ subsumes checking for single faults F_a or F_b.

5 Application: JAS Leakage Detection Subsystem

As a proof of concept we have applied our method to the leakage detection subsystem of the hydraulic system of the JAS 39 Gripen multi-role aircraft, obtained from the Aerospace division of SAAB AB [15]. Both the original system model and our component-based version are described in Esterel [5], a synchronous language whose compiler ensures the nonblocking property upon composition.

5.1 Functionality and Safety

The system's purpose is to detect and stop potential leakages in two hydraulic systems (HS1 and HS2) that provide certain moving parts of the aircraft with mechanical power. Leakages in the hydraulic system could in the worst case lead to such a low hydraulic pressure that the aircraft becomes uncontrollable. To avoid this, four shut-off valves protect some of the branching oil pipes to ensure that at least the other branches keep pressure and supply the moving parts with power if a leakage is detected. However, closing more than one shut-off valve at the same time could result in locking the flight control surfaces and the landing gear which could have disastrous effects. Thus, overall aircraft safety depends on the property φ: *no more than one valve should be closed at the same time.*

5.2 Architectural View

The electronic part of the leakage detection subsystem consists of three electronic components (H-ECU, PLD1 and PLD2), four valves and two sets of sensors (see Figure 2). The H-ECU continually reads the oil reservoir levels of the two hydraulic systems, determines if there is a leakage, and if so, initialises a shut-off sequence of the valves. To ensure that the overall property is satisfied, two programmable logic devices, PLD1 and PLD2, continually read the status of the valves and send signals to them as well. If the readings indicate that more than two valves will close, PLD1 and PLD2 will disallow further closing of any valves. Thus, PLD1 and PLD2 increase the fault tolerance of the shut-off subsystem implemented in the H-ECU.

Each valve is controlled by two electrical signals, one signal on the high side from the PLD2 and one on the low side from the H-ECU. Both of these signals need to be present in order for the valve to close. In this study, we only consider the three components H-ECU, PLD1 and PLD2. Thus, due to the functionality of the valves, the property φ can be replaced by φ': *no more than one valve should receive signals on both the high side and the low side at the same time.*

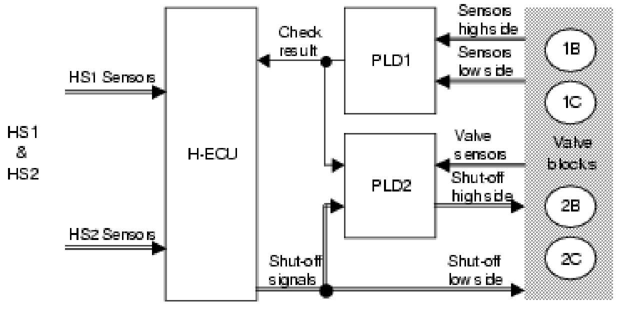


Fig. 2. The hydraulic leakage detection system

5.3 Analysis of Fault Tolerance

Modules: $PLD1$, $PLD2$ and $HECU$ are represented as synchronous modules.
Fault modes: A set of fault modes F_{PLD1}, F_{PLD2} and F_{HECU} for each component has been identified. Every input to the components has been analysed and the possible faults have been modelled as corresponding fault modes.
Safety interface generation: The least restrictive environments E^{φ}_{PLD1}, E^{φ}_{PLD2}, E^{φ}_{HECU} of the components were generated by the algorithm of Section 3.1 using a SAT-based model checker (Prover plugin of the Esterel environment).

The least restrictive environment E^{φ}_{PLD1} of $PLD1$ that makes the system satisfy φ' leaves all the inputs to $PLD1$ unconstrained. By Prop. 2, $PLD1$ in the environment E^{φ}_{PLD1} is also resilient to all faults in F_{PLD1}. Analysis shows that due to their fault-tolerant design, $HECU$ and $PLD2$ satisfy the property φ' with no constraints on their environment whatsoever, i.e., $E^{\varphi}_{PLD2} = E^{\varphi}_{HECU} = True$.

Since none of $E^{\varphi}_{PLD1}, E^{\varphi}_{PLD2}$ and E^{φ}_{HECU} constrain any of the input variables of their corresponding component, these components are resilient to all single faults. Hence, the single fault resilience set of each safety interface will contain every fault mode in the corresponding fault mode set. The generated minimal environments also show that the components are resilient to all double faults, creating a safety interface that includes all pairs of faults in the double fault resilience portion of the safety interface.

Single-component faults: After computing the safety interfaces for the three components in the application (w.r.t. single and double faults), the single component fault analysis becomes trivial. No single or double fault of a single component will cause a threat to system-level safety, since all faults are included in the single fault resilience portion and all pairs of faults are included in the double fault resilience portion of the safety interface.

Multiple-component faults: By checking $\forall j\ M_i \parallel F_k \circ E_i \leq E_j$ for all module-fault pairs (M_i, F_k) where $M_i \in \{PLD1, PLD2, HECU\}$ and $F_k \in F_{PLD1} \cup F_{PLD2} \cup F_{HECU}$ we could conclude that no double fault on input signals would make a threat to system-level safety.

5.4 Results

By generating safety interfaces as described in Section 3 and using the compositional techniques of Section 4 on the aerospace application we concluded that:

- All components in the system are resilient to single faults with respect to the system level safety property φ'.
- All components in the system are resilient to double input value faults with respect to the system level safety property φ'.
- No pair of faults in the system are a threat to system level safety.
- By analysing the components individually and generating the safety interfaces using Propositions 2-4, we were able to perform the fault tolerance analysis *without* composing the whole system.

6 Conclusions

This paper extends component-based development, which has so far focused on efficient system development, to efficient analysis of safety. Certification of safety-critical systems includes providing evidence that a system satisfies certain properties even in presence of undesired faults. This process is especially costly since it has to be repeated for every component change in a system with a long life cycle. We have provided formal models and methods to support this process while hopefully reducing the burden of proof on the system integrator.

Any system will break if there are sufficient numbers of faults in its components at run-time, either due to environment effects or due to inconsistencies in designing interfaces. Safety analyses for industrial products typically assume a number of independent faults and consider the effects of single and potentially double faults. Triple and higher number of faults are typically shown to be unlikely and not studied routinely. Our component interfaces capture what an integrator can assume about the resilience that a component offers with respect to single and double faults. The model could be extended to multiple faults (triple and more) but then the combinatoric complexity would hamper the automatic support for formal analysis. Already with this granularity, we believe that there are enough gains in efficiency for the analysis performed at system level.

This paper uses a general fault model that covers arbitrary value changes at component inputs. While this model is a powerful, for some cases it may be too general and modelling specific faulty behaviour may improve analysis efficiency. The use of reactive modules as generic base allows for more specific models in future studies, such as handling transient faults with given behaviours.

The entire approach has so far had a qualitative nature. Many safety-critical systems have to estimate a quantitative (probabilistic) measure of reliance on a particular analysis. The study of the extensions of this model to quantitative analyses is a topic for future work.

We have assumed that the misbehaviour of a component's environment can be captured by a discrete model (based on value domains with finite range). An extension could consider more specific fault modes arising from interactions with a physical environment given a continuous model. Also common mode failures were excluded at this stage. Another future direction is efficient generation of environment models. The naive approach presented here can be used as a proof of concept and can obviously be improved by more advanced techniques.

Acknowledgements. This work was supported by the Swedish strategic research foundation project (SSF), and the national aerospace research program NFFP. Many thanks are due to our industrial partners from Saab AB for numerous discussions.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
2. O. Åkerlund, S. Nadjm-Tehrani, and G. Stålmarck. Integration of formal methods into system safety and reliability analysis. In *Proceedings of* 17^{th} *International Systems Safety Conference*, pp. 326–336, 1999.
3. R. Alur and T. A. Henzinger. Reactive modules. In *Proc.* 11^{th} *Symposium on Logic in Computer Science*, pp. 207–218. IEEE Computer Society, 1996.
4. U. Aßman. *Invasive Software Composition.* Springer Verlag, 2003.

5. G. Berry. *The Esterel v5 Language Primer.* CMA, Sophia Antipolis, 2000.
6. A. Bondavalli and L. Simoncini. Failures classification with respect to detection. In *Proc. of* 2^{nd} *IEEE Workshop on Future Trends in Distributed Computing Systems*, pp. 47–53. IEEE Computer Society, 1990.
7. M. Bozzano and et al. ESACS: an integrated methodology for design and safety analysis of complex systems. In *ESREL 2003*, pp. 237–245. Balkema, June 2003.
8. A. Burns and A. J. Wellings. HRT-HOOD: a structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, 1994.
9. I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, editors. *Proc. of 7th Int. Symposium on Component-Based Software Engineering*, LNCS 3054. Springer, 2004.
10. F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors. 3^{rd} *Int. Symp. on Formal Methods for Components and Objects*, LNCS. Springer, 2004.
11. J. Elmqvist and S. Nadjm-Tehrani. Intents and upgrades in component-based high-assurance systems. In *Model-driven Software Development.* Springer, 2005.
12. P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey. Towards integrated safety analysis and design. *SIGAPP Applied Computing Review*, 2(1):21–32, 1994.
13. L. Grunske, B. Kaiser, and R. H. Ruessner. Specification and evaluation of safety properties in a component-based software engineering process. In *Embedded system development with components.* Springer Verlag, 2005. to appear.
14. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Proc. AMAST'93*, pp. 83–96. Springer, 1994.
15. J. Hammarberg and S. Nadjm-Tehrani. Formal verification of fault tolerance in safety-critical configurable modules. *International Journal of Software Tools for Technology Transfer*, Online First Issue, Dec. 14, 2004. Springer Verlag.
16. E. Henley and H. Kumamoto. *Reliability Engineering and Risk Assessment.* Prentice Hall, 1981.
17. T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Taşiran. An assume-guarantee rule for checking simulation. In *Proc.* 2^{nd} *Int. Conf. FMCAD*, pp. 421–432, 1998.
18. C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference.* PhD thesis, Oxford University, 1981.
19. J. Jürjens. Developing safety-critical systems with UML. In *Proc.* 6^{th} *Int. Conf. UML 2003*, LNCS 2863, pp. 360–372. Springer, 2003.
20. J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications. *Proc. of the IEEE*, 82(1):25–40, Jan. 1994.
21. H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proc. of the* 17^{th} *IEEE Int. Conf. on Automated Software Engineering*, pp. 195–204. IEEE Computer Society, 2002.
22. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer-Verlag, 1992.
23. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.
24. Y. Papadopoulos, J. A. McDermid, R. Sasse, and G. Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering and System Safety*, 71(3):229–247, 2001.
25. D. J. Pumfrey. *The Principled Design of Computer System Safety Analyses.* PhD thesis, Department of Computer Science, University of York, 2000.
26. A. Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78:1–12, 2002.
27. J. A. Stankovic. Vest - a toolset for constructing and analyzing component based embedded systems. In *Proc. EMSOFT'01*, LNCS 2211, pp. 390–402. Springer, 2001.

28. D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, 1997.
29. E. A. Strunk and J. C. Knight. Assured reconfiguration of embedded real-time software. In *Proc. International Conference on Dependable Systems and Networks*, pp. 367–376. IEEE Computer Society, 2004.
30. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 2nd edition, 2002.
31. A. Tesanovic, S. Nadjm-Tehrani, and J. Hansson. Modular verification of reconfigurable components. In *Embedded System Development with Components*. Springer, 2005. to appear.

A Safety-Related PES for Task-Oriented Real-Time Execution Without Asynchronous Interrupts

Martin Skambraks

Faculty of Electrical and Computer Engineering,
Fernuniversität, 58084 Hagen, Germany
martin.skambraks@fernuni-hagen.de

Abstract. The architectural concept of a safety-related programmable electronic system featuring task-oriented real-time execution is presented. Its most essential characteristics are task execution without the use of asynchronous interrupts, scheduling in direct reference to Universal Time Coordinated, and an integrative hardware approach to detection and processing of failures, forward recovery and non-intrusive monitoring. The architecture is based on physical separation of task execution and task administration, which is realised in form of a digital logic circuit. Time is quantised into Execution Intervals, and tasks are partitioned into Execution Blocks matching these intervals. This concept lowers the complexity of both hardware architecture and temporal behaviour and, thus, conforms particularly well with the safety standard IEC 61508.

1 Introduction

Employing *Programmable Electronic Systems (PESs)* in control applications in which failures can endanger humans or the environment has become quite common in the last two decades. Nevertheless, safety-licensing of these combined software and hardware systems is still problematic. The problems arise less due to inevitable spontaneous physical failures that must be taken into account, as rather from the complexity of such systems, which causes an enormous effort for verification.

The safety standard IEC 61508 limits the complexity indirectly by restricting the use of some conventional processing methods. As an example, its design guidelines only permit limited use of interrupts and pointers in software for the two highest safety integrity levels SIL 3 and SIL 4 (Part 3, Table B.1). For these safety classes, the standard also states that the use of formal methods for software verification as well as the avoidance of dynamic objects and variables is *'Highly Recommended'* (Part 3, Tables A.1 and B.1). The latter term denotes that *'if this technique ... is not used then the rationale behind not using it should be detailed during safety planning and agreed with the assessor'*. These guidelines, which at first glance sound incongruously fuzzy, denote indirectly that *design simplicity* is the key to safety.

The programable systems that are nowadays employed in safety-critical applications follow either the approach of *Periodical Operation* or of *Task Orientation*. The most common representatives of the first category are *Programmable Logic Controllers (PLCs)*. They suit the demands for safety-licensing best because of their inherently simple temporal behaviour. Unfortunately, their field of application is limited to simple control tasks. Task-oriented systems, on the contrary, have a less restricted field of application and a more problem-oriented programming style, but they require far more effort for safety-licensing. But even more significantly, conventional task-based systems use interrupts to control the program flow. This clearly conflicts with the requirements of IEC 61508 for the two highest safety integrity levels.

With the intention to combine the advantages of both the periodic and the task-oriented approach, a novel architectural concept for a real-time PES has been developed. The concept builds on strict separation of task administration, which is realised in form of a digital logic circuit, from the application processor, which executes the application specific software in discrete intervals. This operating principle renders the use of asynchronous interrupts superfluous, and increases the conformity with IEC 61508. As a result, not only the temporal behaviour of the total system, but also the architecture of the embedded processor are simplified. The proposed system incorporates a unified approach for error detection, forward recovery, non-intrusive monitoring and recording of process activities. In comparison to conventional systems, which typically incorporate a combination of several techniques to realise these safety-related functions, this results in a remarkable decrease of system complexity. Instead of processing redundant information inside, the PES is designed to be redundantly configured itself. Non-intrusive monitoring combined with forward recovery of error-affected instances guarantees permanent availability of a redundant configuration.

The main part of this paper is structured as follows. Section 2 categorises conventional PESs into two classes, and discusses their benefits and drawbacks with regard to safety aspects. The task-processing strategy without asynchronous interrupts and its advantages regarding safety-licensing are described in the third section. The fourth one covers feasibility aspects of task-based application software. Section 5 explains the structure and the operating principle of the hardware-implemented task administration. This is followed by some remarks regarding the integration into a holistic safety concept. A short summary at the end recapitulates the most essential aspects, mentions the current state of our work, and states open issues.

2 Categorisation of Existing Systems

The PESs currently employed in safety-critical applications can be categorised into *periodically operating* and *task-oriented* ones, depending on the operating policy they follow. For industrial automation, systems that follow a mixture of both operating principles are available, but this is unfavourable for safety-related technology.

Periodically Operating PESs. This PES class executes application specific programs in processing cycles of constant, fixed duration. It processes the program code always completely within each cycle. The strictly cyclic operating principle facilitates condition-controlled branching merely on a restricted scale; completely process-controlled program flow is not realisable [1,2]. This operating policy does not only limit the field of application to simple control tasks, it also results in a not problem-oriented programming style. Typical representatives of this category are the PLCs, which are mostly programmed following the function block paradigm of IEC 61131-3.

All tasks must be implemented in a way that complies with a 'global' cycle time; individual timing constraints are only considered at second instance. Extensive algorithms either cause long cycle periods, or they must be distributed over consecutive cycles. The former increases the system response time, the latter the complexity of application programs. In addition to that, handling several tasks with extremely different or varying response times is problematic. The program code is typically not processed in direct relation to a global time base, such as Universal Time Co-ordinated (UTC). Hence, additional effort is necessary for synchronisation with external systems and to record all system activities relative to UTC, which is fundamental to enable investigation of cause-and-effect chains in case of nearly simultaneous outages of separate systems.

Nevertheless, the most essential advantage of this PES category is the remarkably low complexity of its hardware architecture and its temporal behaviour. This does not only minimise the effort for safety-licensing, but also makes, in principle, this PES class suitable for applications of the highest safety class (SIL 4). However, the systems currently available off-the-shelf are only certified up to SIL 3, e. g., SIMATIC S7-414H (`www.siemens.com`).

Task-Oriented PESs. This PES class uses interrupts to control software execution, which allows the program flows to be arbitrarily controlled by the processes, and which enables asynchronous processing of several tasks. On the one hand, the asynchronous operating principle increases system complexity, since special mechanisms for task synchronisation like, e. g., semaphores, are necessary. On the other hand, the operating style makes this PES class more flexible and suitable for extensive control applications [3]. Thus, although asynchronous, task-based programming is more problem-oriented than cyclically processed program code, this PES category causes more effort for safety-licensing. This is due to the high complexity of the hardware, the need for a real-time operating system (RTOS), and their interactions with the application software.

A hardware part of significant complexity is the interrupt logic of the processor. Interrupt handling usually involves a (stack) pointer and dynamic memory usage. Thus, the use of interrupts actually contradicts IEC 61508, which restricts the use of pointers and dynamic objects in highly safety-critical applications.

The complexity of the RTOS usually arises from performance reasons. In order to keep the response time of real-time systems short, most RTOSs are subdivided into several layers [4]. The lowest layer serves time-critical functions of low computational extent; more extensive functionalities are processed on higher layers. Minimising the computational load caused by the RTOS kernel functions is another way to keep response times short. That is why mostly priority-based scheduling policies are employed, although time-based strategies, which cause higher computational load, suit the demands of real-time systems better. Provided time-based scheduling is carried out at all, the time representation usually does not comply with the UTC standard. Thus, recording all system activities in reference to UTC requires conversion of system time to UTC. The necessity of such conversions as well as the layered structure mentioned above increase the complexity of RTOSs considerably.

The complexity caused by the interaction of processor, RTOS and application software results mainly from the dependence of execution times on the process. For this reason, *proving temporal feasibility* of an application software is a fundamental part of its development. Not only the use of mechanisms for task synchronisation causes difficulties for feasibility analysis, but also the fact that each interrupt induces a context-switch of the processor which, in turn, influences the response times of all activated tasks. The distinction between interruptible and non-interruptible program parts further aggravates the situation.

Although the use of interrupts does not perfectly comply with IEC 61508 for the two highest safety integrity levels, there are task-based real-time operating systems available that are certified for SIL 3, e. g., OSE RTOS (`ww.ose.com`).

3 Task Execution Without Asynchronous Interrupts

The concept introduced here combines the advantages of both PES categories discussed above. This is realised by physical separation of *task execution* and *task administration*. A Task Processing Unit (TPU) executes application specific program code. Due to the beneficial characteristics in terms of safety and security, it contains a processor with the Harvard architecture. The Task Administration Unit (TAU) is responsible for task state transitions and processor scheduling. For the latter the policy Earliest-Deadline-First (EDF) is used. Several proofs exist that this strategy always leads to feasible schedules, provided timely execution is possible at all, e. g., [5,6].

Time is quantised into discrete *Execution Intervals*, and tasks are *partitioned* into a number of *Execution Blocks* each. The Execution Intervals have a fixed duration, and are defined for the physically separated TAU and TPU which operate cyclically and in synchrony. They have the following characteristics.

- Each Execution Block is executable within a single Execution Interval.
- The execution of a block is not pre-emptable.
- Data exchange between blocks is only possible via the TPU memory; the content of the processor registers is lost at the end of each interval.
- The Execution Blocks of a task are indexed for identification.
- The Execution Blocks of a task do not need to be executed in consecutive order. For each task, the TAU stores a parameter called *NextBlock* in the Task List, which identifies the subsequent Execution Block.

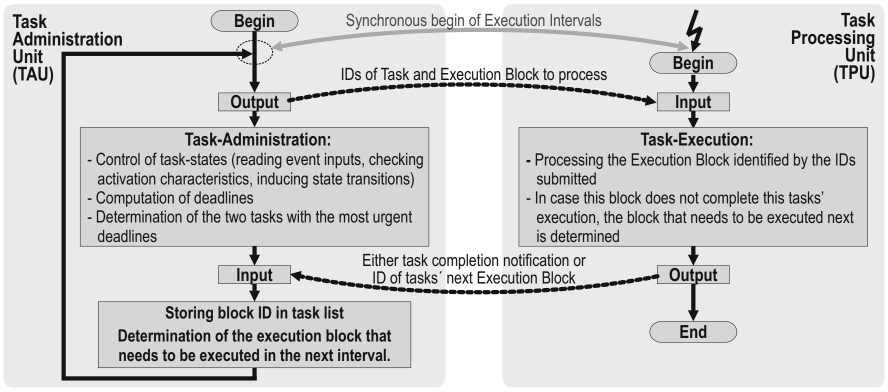

Task Administration Unit (TAU)
Begin
Synchronous begin of Execution Intervals
Task Processing Unit (TPU)
Begin
IDs of Task and Execution Block to process
Output
Input
Task-Administration:
- Control of task-states (reading event inputs, checking activation characteristics, inducing state transitions)
- Computation of deadlines
- Determination of the two tasks with the most urgent deadlines
Task-Execution:
- Processing the Execution Block identified by the IDs submitted
- In case this block does not complete this tasks' execution, the block that needs to be executed next is determined
Either task completion notification or ID of tasks´ next Execution Block
Input
Output
Storing block ID in task list
Determination of the execution block that needs to be executed in the next interval.
End

Fig. 1. Illustration of the operating principle without interrupts

The operating principle can be roughly described as follows: At the beginning of each Execution Interval, the TAU outputs the *ID of Block to Execute*, which identifies the next Execution Block of the task that must be executed according to the scheduling algorithm. The ID corresponds to the task's *NextBlock* parameter stored in the task list. After the TPU has read this ID, it processes the associated Execution Block. When the APU completes the block at the end of the Execution Interval, it outputs the *ID of Next Block* identifying the task's Execution Block that needs to be executed next. The TAU reads the ID and stores it in the task list as new NextBlock parameter. The flow chart in Fig. 1 illustrates this mode of operation in more detail.

If the executed block was a task's last one, i. e., if a task has been executed completely, the APU outputs the block ID 'Nil'. This completion is taken into account when the TAU determines the *ID of Block to Execute* for the next Execution Interval. That is why the TAU – while the TPU processes an Execution Block – does not only determine the task with the earliest, but also the task with the earliest-but-one deadline. This enables the TAU to immediately output the NextBlock identifier of the task with the next-but-one deadline, in case the task with the next deadline corresponds to the task just been processed *and* just been completed by the APU. The actual state transfer that causes the completed task to be no longer in the state *Activated* is carried out in the subsequent interval.

This operating principle renouncing the use of asynchronous interrupts makes synchronisation mechanisms such as semaphores superfluous, since any task has non-interrupted exclusive access to the processor during an Execution Interval. Tasks can communicate with each other via the data memory without the danger of interruptions while writing messages. Using variables stored in the data memory, mutually exclusive access to peripheral components can be realised by simple means. Alternating access of several tasks can be realised in a similar way, it only requires the capability to induce task suspension and continuation by program. The absence of special synchronisation mechanisms results in further design simplification.

Typically, the parts of a processor supporting interrupts are the most complex ones, and render formal verification to be either unacceptably expensive or even impossible. Thus, task execution without asynchronous interrupts also minimises the complexity of the TPU processor, as no interrupt logic is necessary. This simplification eases proving correctness with formal means. Moreover, since IEC 61508 restricts the use of pointers, dynamic objects and interrupts in highly safety-critical applications, the proposed PES concept complies better with the safety standard than conventional task-oriented systems.

In summary, this concept of task execution without the necessity of asynchronous interrupts significantly simplifies the temporal behaviour as well as the hardware structure, eases formal verification, and increases conformity with IEC 61508 for systems of highest safety criticality (SIL 3/SIL 4). Of course, the proposed operating principle requires special compilation of the application software.

4 Schedulability Aspect

The proposed operating principle does not restrict the field of application as the paradigm of periodical operation does. Without the use of asynchronous interrupts, it features completely process-dependent program flows. As for any task-based real-time system, this capability of arbitrary program flows necessitates to check whether the application software can run in a timely fashion under any circumstances.

Intended to ease this feasibility analysis, the proposed concept supports a particularly simple task state model on the lowest possible level – the hardware level. The model bases on three characteristics: *Worst Case Execution Time* t_C, *Maximum Response Time* t_B, and *Minimum Activation Period* t_T. Following the definitions in [7], these *Execution Characteristics* allow to specify the temporal behaviour of any hybrid task set, consisting of both periodic and sporadic tasks. Fig. 2 illustrates the model. It differs from other ones (cf., e. g., [8]) by the state *Suppressed*.

The Minimum Activation Period equals the minimum duration between two consecutive activations of the same task. Thus, this parameter limits the computational load indirectly. Only tasks in the state *Known* can be activated. In case

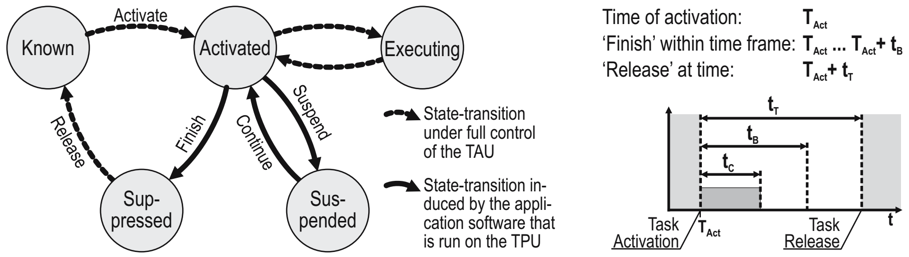

Known
Activate
Activated
Executing
Suspend
Finish
Continue
Release
Sup-pressed
Sus-pended
State-transition under full control of the TAU
State-transition induced by the application software that is run on the TPU
Time of activation: T_{Act}
'Finish' within time frame: T_{Act} ... T_{Act}+ t_B
'Release' at time: T_{Act}+ t_T
t_T
t_B
t_C
Task Activation
T_{Act}
Task Release
t

Fig. 2. The applied task state model

a task is completely processed before the time frame $T_{Act} \ldots T_{Act} + t_T$ elapses, it is transferred to the state *Suppressed*. The transition back to the state *Known* is carried out at $T_{Act} + t_T$, enabling further activations. The state *Suspended* is provided for synchronisation purposes.

All transitions are under full control of the TAU, except the transitions *Finish*, *Suspend* and *Continue*, which are performed by the TAU, but induced by the application software. A task automatically induces the state transition *Finish* when its execution completes. Since the application software is processed in non-pre-emptable intervals, state transitions can occur at interval ends, only. Thus, in case a task induces the state transition *Suspend* by its program, the suspension will take place at the end of the interval. A task can only suspend itself, but it can induce *Continue* transitions for all other tasks. In combination with the cyclic operating policy, this allows to enforce precedence relations between tasks, which are necessary to realise, e. g., alternating access.

Since the task state model is supported by hardware, the activation of a task is restricted to one instance at a time on the lowest possible level. This complies perfectly well with the safety standard IEC 61508, which prohibits dynamic instantiation of objects for applications of highest safety criticality.

Taking this behaviour into account, the hardware-supported *Execution Characteristics* t_C, t_B and t_T allow to formally prove the feasibility of application software. Suitable feasibility conditions have already been developed and are discussed in various publications, e. g., [8,9,10,11]. A comprehensive presentation of appropriate analysis methods can be found in [7].

5 Hardware-Implemented Real-Time Scheduler

In contrast to various other real-time systems, the PES concept introduced here processes *all* internal actions in reference to UTC, which is the internationally standardised sole *legal* time reference. Appropriate synchronisation signals are available worldwide via, for instance, GPS, GLONASS and – in the future – GNSS. The synchronisation to global time reduces the complexity of distributed systems significantly, since the problems related to varying time bases are prevented. Of course, deviations from UTC (e. g., due to temporary reception outages) need to be taken into account.

The TAU follows the task scheduling concept of the **P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage (PEARL), which was standardised in DIN 66253-2. One of the interesting features of this language is its direct notion of time [12]. This enables exact and problem-oriented specification of temporal conditions to activate, terminate, suspend, continue or resume tasks [13]. A periodic task activation during a given time-frame is specified by

```
AT {clock-expression | [asynchronous-event-expression] + duration1}
EVERY duration2 DURING duration3 ACTIVATE task-name.
```

This is the most general form of a task activation schedule [8]. The hardware implementation of the TAU inherently supports such activation plans. Therefore, each task is assigned a set of parameters, which facilitates configuration for various different activation conditions, e. g., at a predefined delay after the occurrence of an asynchronous signal.

Most real-time systems follow the approach of rate-monotonic, fixed-priority scheduling (cf., [14]). This 'timeless' scheduling policy is merely advantageous for systems that do not provide explicit support for timing constraints, such as periods and deadlines [15]. Thus, rate-monotonic scheduling is actually inadequate for safety-related systems, for which hard real-time constraints always *must* be specified exactly. It potentially causes unnecessary task switches, since only the current system state is taken into account to make resource assignments; the instant of a task's activation is neglected. The proposed PES implements the EDF algorithm in hardware, rendering the differences between priority-based and EDF scheduling to be irrelevant in terms of computing time. Thus, there is no need for task priorities, and the time-based scheduling policy can be applied.

A perfect real-time operating system would permanently check the activation conditions of all tasks, and uninterruptedly inform about the task to be executed according to the scheduling policy. This demand for a continuous working pattern leads to the objective of implementing the TAU in form of a digital logic circuit that processes the kernel algorithms for all tasks in parallel. Unfortunately, considering the EDF algorithm and the fact, that a typical real-time application consists of some 10 to 50 tasks, it is obvious that a completely parallel operating logic circuit implementation of the TAU would require an unacceptably high amount of logic gates.

For this reason, the hardware architecture of the TAU combines parallel and sequential processing. The kernel algorithms are structured in a way as to allow for parallel processing of the operations related to a single task, whereas all tasks are sequentially subjected to these operations. Fig. 3 illustrates this processing pattern. It shows the main parts of the Task Administration Unit (TAU), viz., *Task Data Administration (TDA)* and the unit for *Activation Control and Scheduling (ACS).*

The TDA administrates a *Task List*, which contains a set of parameters for each task such as its current state and its execution characteristics. The

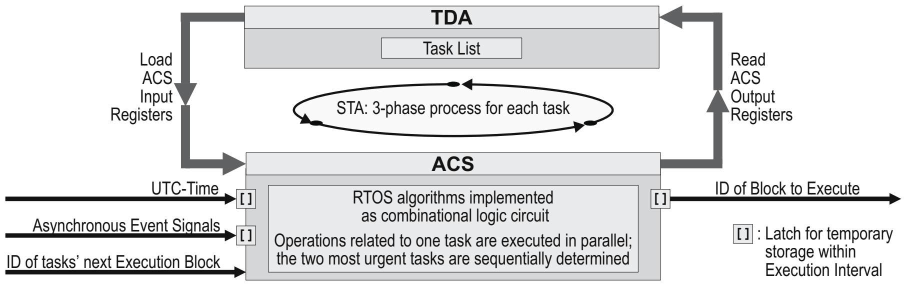

TDA
Task List
Load ACS Input Registers
Read ACS Output Registers
STA: 3-phase process for each task
ACS
UTC-Time
Asynchronous Event Signals
ID of tasks' next Execution Block
RTOS algorithms implemented as combinational logic circuit
Operations related to one task are executed in parallel; the two most urgent tasks are sequentially determined
ID of Block to Execute
[ ] : Latch for temporary storage within Execution Interval

Fig. 3. The STA consists of the TDA and the ACS. While they do the Sequential Task Administration (STA), a 3-phase process is carried out once for each task.

task list has a static size, i. e., all tasks a certain application is consisting of must be registered with the TAU at set-up time. Thus, in conformance with the requirements of IEC 61508 for SIL 4 applications, dynamic instantiation of tasks (resp. objects) is not supported. Instead, the activation characteristics of a task can be modified.

The TDA co-operates closely with the ACS while sequentially processing all tasks within each Execution Interval. During this *Sequential Task Administration (STA)*, the TDA initiates for each task a three-phase process:

1. First, the TDA accesses the Task List and transfers the task's entire data set to dedicated input registers of the ACS.
2. Then, the ACS processes the task data and outputs an updated data set. This is done by combinational logic within one clock cycle.
3. During the last phase, the TDA transfers the updated task data from the ACS back to the Task List.

This way, the ACS carries out the following operations in the course of the STA:

1. Checking the activation characteristics (e. g., checking time schedules or asynchronous occurrences)
2. Supervising task state transitions
3. Computing deadlines
4. Generating updated task parameters
5. Identifying the task with the earliest deadline and the one next in line
6. Output of the *ID of Block to Execute*

The first four operations are separately executable for each task. Therefore, they are performed in parallel by a combinational digital circuit. The fifth item requires comparing the deadlines of all activated tasks. This is carried out sequentially, while the IDs of the two most urgent tasks are temporarily stored within the iterations of the 3-phase process. The ID of the Execution Block that needs to be processed in the subsequent interval is output at the end of an Execution Interval, after the TPU submitted an ID to the TAU.

Since the TDA and the ACS are implementable in form of a digital logic circuit, extremely short response times are realisable without the use of a multi-layered operating system structure and without minimising the computational effort by applying a priority-based scheduling algorithm, as it is usually done in conventional real-time systems. Therefore, the proposed hardware scheduling concept has a significant lower complexity than a conventional task-oriented real-time operating system with an equally short response time.

6 Integration into a Holistic Safety Concept

Hitherto, merely the avoidance of design faults through *Design Simplicity* has been considered. In order to guarantee safe and reliable operation, a holistic safety concept is necessary, which also takes spontaneous hardware failures into account, but does not ruin simplicity. This is achieved by exchanging *Serial Data Streams* via a particularly tightly integrated *Communication Interface.*

The Concept of Serial Data Streams. IEC 61508 recommends various techniques to reduce the influence of hardware failures. Some of them are, e. g., redundant memory banks or multiple processors combined by majority voting. However, most techniques take few failure sources into account, only, and a combination of several techniques is necessary to cope with all possible failures. Thus, a single approach covering all failure sources is desirable in order to minimise system complexity.

That is why a rather exceptional but more integrative approach was taken to reduce the influence of hardware failures. Instead of processing redundant information inside, the PES itself is designed to be redundantly configured. Each PES instance outputs a *Serial Data Stream (SDS)* that provides full information about internal processing. In a redundant configuration, these SDSs are exchanged between PES instances to serve the following four safety functions.

Non-intrusive monitoring: Since the SDSs inform about the internal processing, they enable non-intrusive monitoring by external devices.

Recording process activities: The SDSs can be utilised to externally record the system behaviour for later program flow analysis.

Detecting processing errors: Each PES can detect processing errors by comparing its SDS with the SDSs of other PESs.

Forward recovery at runtime: In case a PES is affected by a transient hardware fault, the SDSs of redundant PESs enable to copy the internal state and to resume processing at runtime.

The SDS concept bases on the fact that the maximum amount of data changes inside a PES is indirectly limited by the execution characteristics discussed in Section 4. The SDS is organised in transfer cycles that match the Execution Intervals. Within each cycle, the SDS transfers information about a fixed number of internal data changes. This number sets the limit of data changes permitted.

Data changes can be categorised into *TAU Data Changes (TAUDCs)* and *TPU Data Changes (TPUDCs)*. The frequency of TPUDCs is limited by restricting the amount of an Execution Block's write accesses to a number that is transferable via SDS within each transfer cycle. The number of TAUDCs depends on the frequency of task state changes and modifications of the task parameters (e. g., activation characteristics). This number is directly limited by the execution characteristics of all tasks. In contrast to the TPUDCs, the TAUDCs of one Execution Interval are not necessarily transferable within one cycle. This is because, in theory, it is possible that all tasks perform a state transition simultaneously. This would cause a huge amount of data changes inside the TAU (e. g., storing the activation time of each task), and the capability to transfer them within each Execution Interval would require an undesirably high bandwidth. That is why an integer number is assigned to each TAU data word that represents the *Age* of the stored value. By default, each Age parameter is set to '0'. Any time a data word is modified, the associated Age integer is set to the maximum representable value. If an Age value does not equal '0' or '1', it is decremented by one at the beginning of every Execution Interval. Thus, the lowest integer values (except '0') identify the 'oldest' modified data words. By restricting to the oldest data words, the amount of data that must be transferred within each interval can be bounded.

The previous paragraph describes only how the transfer of *data changes* inside the TAU and the TPU is realised via SDS. In order to enable copying the internal PES state completely by observing its SDS over a pre-defined period of time, it is also necessary to transfer the data that were not modified recently (e. g. RAM data words that were written a few minutes ago). A small extension of the concepts described above realises this *complete* state transfer. Fig. 4 illustrates the data transfer via SDS.

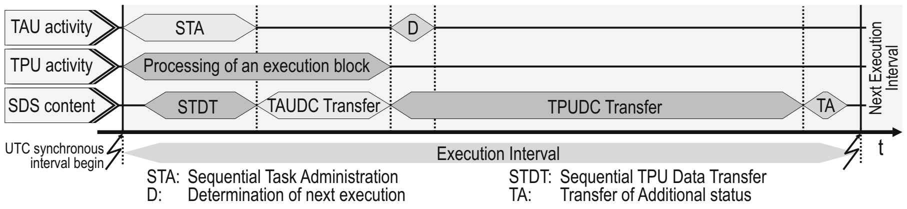

TAU activity
TPU activity
SDS content
STA
D
Processing of an execution block
STDT
TAUDC Transfer
TPUDC Transfer
TA
Next Execution Interval
UTC synchronous interval begin
Execution Interval
t
STA: Sequential Task Administration
D: Determination of next execution
STDT: Sequential TPU Data Transfer
TA: Transfer of Additional status

Fig. 4. Illustration of the data transfer via SDS

While the TAU performs the STA, the SDS transfers a subset of the TPU's RAM content. The entire RAM content is transferred within a number of consecutive Execution Intervals, and in synchrony with UTC. When the STA has been finished, the TAUDCs are transferred. Therefore, the TAU inserts a subset of its oldest data words in the SDS and sets the associated Age integers to '0'. Complete transfer of the TAU data is achieved by changing the 'Ages' of all data words at UTC-synchronous instants. After the TPU completes the current Execution Block, the SDS transfers all its write accesses. Finally, some additional status information like, e. g., the *ID of Block to Execute* is transferred via SDS.

By observing the SDSs of redundant PES instances, this technique does not only allow to detect errors simply by majority voting, but also to copy the internal state from one PES to another at runtime. Thus, in case a PES is affected by a transient failure, it can resume its processing by evaluating the SDSs of the PES instances that remain running. Realising this technique of *Forward Recovery at Runtime* without affecting the minimum achievable response time was only possible by implementing its algorithms – together with the TAU – as digital logic circuit.

Obviously, the performance achievable with the proposed PES concept is strongly limited by the data transfer bandwidth of the communication interface. However, computational performance is not the major concern for safety-related systems, and the high bandwidths of modern transfer technologies allow for a performance more than sufficient for most safety-critical applications.

Communication Interface. For communication *one* interface is used which does not only suit the needs to interconnect multiple PES instances of a redundant configuration, but also supports data exchange with the process peripherals

(e. g., sensors, actuators). This results in low wiring expenses. The communication technique bases on the field bus ‘Interbus S’, because of its low hardware requirements and inherent simplicity [16]. All system nodes, i. e., the redundant PES instances as well as sensors and actuators, are connected to a ring, and data are transferred from node to node as in a shift register. Like the SDS transfer, the I/O data transfer is cyclic: Before a data word is accessible for further processing, it passes all nodes within one Execution Interval.

Conventional ring-based communication techniques increase safety by using both possible transfer directions. This approach cannot guarantee system availability in case of more than one ring interruption or device outage. That is why the proposed PES follows a different approach, which is illustrated in Fig. 5.

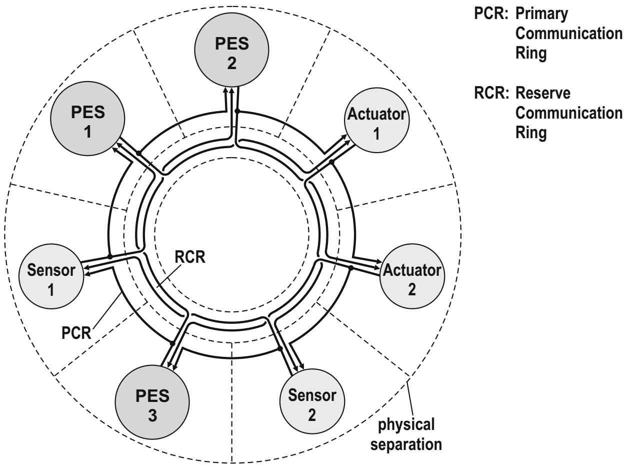


Fig. 5. Illustration of the multi-ring connection scheme with three redundant PESs

Each node’s outgoing interface is connected via multiple, physically separated communication rings to multiple recipients. The first communication ring defines the ring order, the second communication ring directly connects nodes which are ordered as next-but-one in the first ring, and so on. This ‘multi-ring’ technique allows scalable safety by adding further rings and, moreover, can even be combined with the bi-directional method.

7 Conclusion

A novel architectural concept for safety-related PESs was designed following the policy ‘Progress is the road from the primitive via the complicated to the simple’, stated by Biedenkopf in 1994 [17]. Both the PES’s architecture as well as its operating principle are of remarkable simplicity, which eases verification and – even more importantly – lowers the cost for safety-licensing.

Simplicity of design is achieved by physically separating task administration and task execution. The application software is organised in tasks, but – contrary to conventional task-oriented systems – the tasks are subdivided into a number of blocks which are executed in discrete intervals of constant duration. This operating principle renders the use of asynchronous interrupts superfluous. As a result, not only the hardware architecture is simplified, but also the temporal behaviour of the total system. Task administration is carried out by a special-purpose logic circuit. This enables task scheduling in direct relation to Universal Time Co-ordinated. Moreover, the hardware-implemented task administration ensures short response times without utilising a complex multi-layered structure nor minimising the computational effort by applying a primitive and inadequate priority-based scheduling algorithm. Instead, tasks are scheduled according to the time-based Earliest-Deadline-First policy.

Many techniques were developed in the past to increase the reliability and availability of programmable electronic systems. Unfortunately, most of them cover a small amount of failure sources, only, and various techniques need to be combined to achieve safety, causing design complexity to increase significantly. The proposed system integrates error detection, forward recovery and non-intrusive monitoring in a unified way. This functional unification, which bases on exchanging Serial Data Streams between redundant PES instances, results in a hardware design of minimum complexity and especially low cost. Moreover, it combines the capability of forward recovery at runtime with an remarkably low achievable minimum response time.

The proposed PES concept is completed by an inherently simple communication strategy which serves data exchange between redundant PES instances and with the process peripherals. A 'multi-ring' technique combines high, scalable reliability and low wiring expenses with an especially simple structure. Like the application processing, communication is performed in discrete intervals, too.

All system components operate cyclically and in synchrony, leading to a simple and easy-to-model temporal behaviour of the entire system. As a result, formal verification of application-specific software is simplified. In comparison to conventional periodically operating PESs, the proposed PES's field of application is larger, since it is capable of task-based software execution with process-dependent program flows.

So far, a VHDL description has been prepared that realises the proposed PES concept as a System-on-Chip. The TPU is based on a soft-processor version of the well-known processor Intel 8051, which is provided by Oregano Systems. The VHDL design has extensively been tested by simulation and, then, implemented in an FPGA. A software that automatically divides the task's program code into Execution Blocks has also been created. Unfortunately, the splitting of program code into blocks could not be discussed in this document due to page limitations. Currently, we investigate performance aspects of subdividing software into Execution Blocks to find the optimum length for Execution Intervals.

Throughout our work, we considered 'Design for simplicity' as a key factor to achieve safety. The question remaining open is: What characteristics make a design simple? Obviously, design simplicity conflicts with computing performance and resource efficiency, just as these two design constraints conflict with each other. But in times in which 'dealing with complexity of large systems' is considered as a major problem, simplicity is of equal importance. Thus, our future work focuses on the questions: What general guidelines can be used to simplify the design of a safety-related real-time system? How do they affect the operating principle, the computing performance, and the resource efficiency? What measures should be taken into account in order to find an optimum balance between these design constraints?

This article covered only the fundamental concepts of the proposed PES architecture; various aspects could not be discussed due to page limitations. The interested reader is invited to contact the author for further information.

References

1. Bolton, W.: Programmable Logic Controllers. Elsevier Books, Oxford (2003)
2. Rabiee, M.: Programmable Logic Controllers: Hardware and Programming. Ingram, New Orleans (2002)
3. Shaw, A.C.: Real-Time Systems and Software. John Wiley, New York (2001)
4. VDI: Richtlinie VDI/VDE 3554: Funktionelle Beschreibung von Prozessrechner-Betriebssystemen. Beuth Verlag, Berlin-Cologne (1982)
5. Dertouzos, M.L.: Control robotics: the procedural control of physical processes. Information Processing **74** (1974)
6. Henn, R.: Feasible processor allocation in a hard-real-time environment. Real-Time Systems **1** (1989) 77 – 93
7. Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo, G.C.: Deadline Scheduling for Real-Time Sytems, EDF and Related Algorithms. Kluwer Academic Publishers, Boston (1998)
8. Halang, W.A., Stoyenko, A.D.: Constructing Predictable Real Time Systems. Kluwer Academic Publishers, Boston (1991)
9. Spuri, M.: Analysis of deadline scheduled real-time systems. In: Rapport the Recherche RR-2873, Le Chesnay Cedex, France, INRIA (1996)
10. Buttazzo, G.C.: Hard Real Time Systems, Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, Boston (2002)
11. Sha, L., Abdelzaher, T., Årzén, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real-time scheduling theory: A historical perspective. Real-Time Systems **28** (2004) 101–155
12. Hamuda, G., Tsai, G.: Formal specification of a real-time operating systems's component. In: Real-Time Programming 2003, WRTP Conference Proceedings, Elsevier Science Ltd (2003)
13. GI: PEARL90 Language Report. 2.2. edn. Technical Commitee 4.4.2 (Realtime programming, PEARL), Bonn (1998)
14. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. 3rd edn. Pearson Addison Wesley, Harlow, England (2001)
15. Buttazzo, G.C.: Rate monotonic vs. edf: Judgement day. Real-Time Systems **29** (2005) 5–26
16. Baginski, A., Müller, M.: Interbus. Hüthig Verlag, Heidelberg (1998)
17. Biedenkopf, K.: Komplexität und kompliziertheit (complexity and complicateness). Informatik Spektrum **17** (1994) 82–86

Are High-Level Languages Suitable for Robust Telecoms Software?*

J.H. Nyström[1], P.W. Trinder[1], and D.J. King[2]

[1] School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, EH14 4AS, Scotland
{jann, trinder}@macs.hw.ac.uk
[2] UK Software & Systems Engineering Research Group,
Motorola Labs, Basingstoke, England
David.King@motorola.com

Abstract. In the telecommunications sector product development must minimise time to market while delivering high levels of dependability, availability, maintainability and scalability. High level languages are concise and hence potentially enable the fast production of maintainable software. This paper investigates the potential of one such language, Erlang, to deliver robust distributed telecoms software. The evaluation is based on a typical non-trivial distributed telecoms application, a Dispatch Call Controller (DCC) measured on a Beowulf cluster. Our investigations show that the Erlang implementation meets the DCC's resource reclamation and soft real-time requirements, before focusing on the following reliability properties.

- **Availability**, e.g. recovery from failures is fast and repeated failures don't reduce post-recovery throughput.
- **Redundancy degree**, e.g. how many simultaneous copies of the system state can be maintained without impairing performance?
- **Resilience**, e.g. achieving a throughput of 101% at 1000% load on 4 processors.
- **Dynamic adaptability**, e.g. the system can be dynamically upgraded by adding nodes without interruption of service.

We critique Erlang's fault tolerance model, arguing that it is low cost, parameterizable and generic. As the Erlang DCC is less than a quarter of the size of a similar C++/CORBA implementation, the product development in Erlang should be fast, and the code maintainable. We conclude that Erlang and associated libraries are suitable for the rapid development of maintainable and highly reliable distributed products.

1 Challenges for Distributed Telecoms Software

The demands of today's telecoms consumer are simply put: they want low-cost, easy to use systems that are reliable and always available. Achieving such high

* The project is in part funded by the UK EPSRC Research Project, GR/R88137 and is a joint venture between Motorola Software and Systems Engineering Research Group at Basingstoke (England) and Heriot-Watt University at Edinburgh(Scotland).

standards of reliability and availability presents a grand challenge to the telecoms industry, especially when the architecture by necessity is distributed, uses an array of different hardware, networks, operating systems, as well as application software. In the face of stiff competition, reducing time-to-market is essential.

Currently many telecoms systems are implemented in C/SDL on real-time operating systems with trends towards using C++/CORBA, JAVA/RMI and UML 2.0 statemachines. Higher level programming language technologies like Erlang [1] potentially offer significant advantages. Development time can be reduced, and maintainability improved because programs are shorter as they specify less low-level detail. Reliability is improved by safe type systems and relatively easy verification, using an interactive proof assistant with an embedding of the language in the proof rules [2]. Clearly the language technology used must also meet the other functional requirements of telecommunication applications, e.g. real-time requirements.

This paper reports the results of an investigation into the effectiveness of Erlang for constructing robust telecoms software. We start by giving a brief introduction to Erlang (Section 2). As a basis for the evaluation we have re-engineered a substantial Dispatch Call Controller (DCC), originally 21K lines of C++/CORBA, as 5k lines of Erlang (Section 3). We show that the Erlang implementation meets the resource reclamation and soft real-time requirements of the DCC.

The primary focus of our investigation is reliability. Availability experiments investigate single, repeated and multiple failures of various system components (Section 4.1). Redundancy experiments investigate how many workers with fully replicated system state can be maintained (Section 4.2). Resilience experiments investigate performance under 100%, 200% and 1000% load (Section 4.3). Dynamic adaptability experiments investigate system throughput as processors are removed and added (Section 4.5). To draw general conclusions about the suitability of Erlang for constructing robust software we critique its fault tolerance model, using illustrations drawn from the DCC (Section 5).

2 Erlang

Erlang is a distributed functional language developed specifically for constructing high reliability telecommunications systems [3]. It is a new software development technology seeking to establish itself in the telecommunications area. To reduce time to market the language is supplied with an Open Telecom Platform (OTP) [4] that includes inter alia: libraries, tools, distributed debugging support, foreign language interfaces and design principles. Erlang is open-source and supported by commercial training courses and technical support, an international user conference together with books and online reference material.

Erlang has been used by a number of large telecommunications companies to construct a wide range of such applications as the first implementation of GPRS for standard packet data in GSM systems [5], and the Intelligent Network Service Creation Environment [6]. The largest application to date is the AXD

301 scalable and robust backbone ATM switch [7], currently utilising up to 32 Processing Elements (PEs). The code comprises over 1.7 Million lines of new Erlang code [1, p.6], 300K lines of mostly-reused C and 8K lines of Java, developed by a team peaking at 50 software engineers [8].

Erlang has several features that facilitate the construction of large distributed real-time systems. The module system allows the structuring of very large programs into conceptually manageable units. Reliable software is constructed using fault tolerance mechanisms, namely timeouts, exceptions and a mechanism where a process can monitor the termination of other processes. Code loading primitives allow the system to be upgraded without stopping the system: an essential requirement for many applications including telephone exchanges. Erlang has a process-based model of concurrency, and processes communicate by message-passing. Erlang supports single-assignment variables, and has an explicit notion of time, enabling it to support soft real-time applications, i.e. where response times are in the order of milliseconds.

3 Dispatch Call Controller (DCC) and Platform

To evaluate Erlang we have chosen to re-engineer an existing prototype dispatch call system developed at Motorola Labs, Illinois [9]. Dispatch call processing is a prevalent feature of many wireless communication systems. Managing the call processing with a distributed paradigm enables the processing to be scaled as system usage grows, with work dynamically distributed to the resources available. The essence of the application is a group call manager that generates instances of a group call factory dynamically on the resources available. Each factory generates call handlers to manage individual calls sent to it by the manager. Both C++/CORBA and Java/JINI DCC implementations exist [9].

The DCC requires the following functionalities. Dynamic scalability, i.e. the ability to adapt to use additional resources while the system is running. Resource reclamation, ensuring that once a service instance has terminated, its resources are reclaimed. Fault tolerance and providing continued service despite failures in particular. Soft real time performance: call management mustn't interrupt the call.

3.1 The DCC Model

The requirements for the DCC model are derived from the technical report by Lillie [9] and on the functional requirements document from that project [10].

The experiments have been conducted using a model of the DCC service that only deals with regular voice point-to-point calls and hand-offs between the base radio stations. Calls received when the workers are already executing the maximum number of instances, are rejected.

The system may have between 1 and 4 worker nodes running on separate processing elements and 13 processing elements dedicated to the traffic generator and system interface.

The model of the DCC service comprises two distinct parts, the subscribers of the system generating traffic and the fixed-end terminating the service.

The subscribers are simulated by two processes each; the first will issue hand-off messages to the fixed-end notifying it that the subscriber is now associated with a new base radio station, thereby simulating the roaming of the subscriber. The second process setups and generates the voice traffic for the call. The call is initiated by a request from the caller process which waits for the fixed-end setting up the call and routing the traffic; a timer is set when the request is sent modelling that a user would terminate the call if the call is not commenced within a short timespan.

The rates of hand-offs and call requests are higher than one would normally expect from a real system, this is in order to generate large enough volumes of traffic without having an unwieldy number of subscribers in the generator. The requests and the duration of calls are evenly distributed over the expected rates which are: one hand-over every two minutes; one call request every ten minutes; each call last half a minute. During a call the initiating party, which is also the sending party this being simplex traffic, will issue voice data each 90 milliseconds.

The fixed-end is simulated by the framework with a service dealing with both hand-offs and call requests. The service requires one database table to keep track of which subscribers are associated with which base radio station. The hand-offs only result in that the record keeping track of the subscriber is updated; whereas for the calls the service instance will be maintained as a handler for the duration of the call. The handler is maintained for a certain time after the call is completed, should a new call be setup for the same subscriber within a short timespan. The call handler in the fixed-end must also be able to deal with hand-offs during the call.

3.2 Platform

The hardware platform is a 32-node Beowulf cluster, where each node consists of a Pentium-III 530 MHz processor with 256 Mbytes of RAM, interconnected by a switched 100Mbit Ethernet.

The software test platform is made up of a number of subsystems described below and the overall architecture is shown in Figure 1.

Test Management. Responsible for starting and controlling the System Management and Traffic Generator subsystems during the test. The Test Manager will also inject the faults into the non-testing subsystems.

Traffic Generator. The Traffic Generator sends a sequence of calls to the Service Port and acts as a sink for all messages from service instances to caller.

System Management. Responsible for starting, stopping and management of the Service Port and the worker nodes. The System Management subsystem is also responsible for restarting the Service Port for any worker that fails.

Service Port. The Port is responsible for starting and maintaining all the interfaces used by the services to communicate with the Workers and relays calls

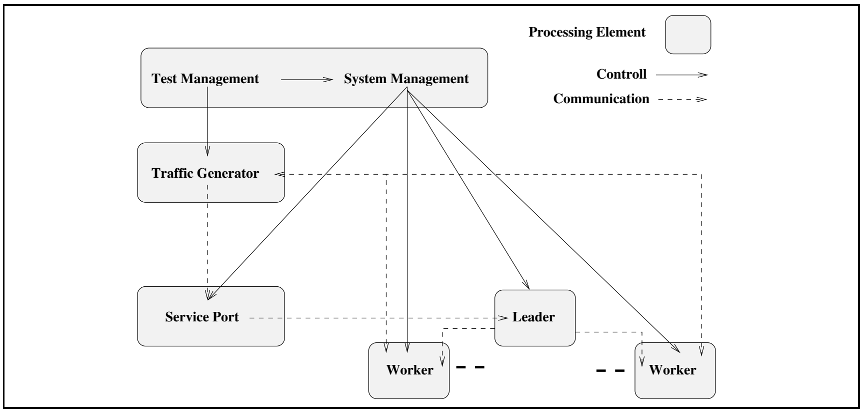


Fig. 1. System architecture

from the subscribers of the services to the Worker responsible for Service Admission acting as gatekeeper.

Worker. There is one or more Worker subsystems in the system and they are responsible for executing of the dispatch call handlers. One of the workers is the designated leader of the workers and is responsible for admission control and distribution of calls between the available Workers. The leader is also responsible for redistributing the call handled by a worker should it fail and for restarting the System Management subsystem should it fail. Should there only be the leader it will also act as a normal worker.

The leader of the workers is selected using a distributed leader election protocol based on functionality in a library enabling distributed name registration. Should the Leader fail a new leader will be elected using the same protocol.

4 Experiments

This section reports investigations of robustness properties of the DCC, specifically availability, redundancy, resilience, telecoms characteristics and dynamic adaptability.

4.1 Availability

An important property of any distributed system, that is expected to have a high availability, is that not only should it be able to handle failure of one of the component systems but also repeated and simultaneous failure in several components.

The reason that it is important that the system can deal with multiple components failing is that it is not uncommon that the behaviour of a faulty component may well cause other components to fail before it fails itself; even if the

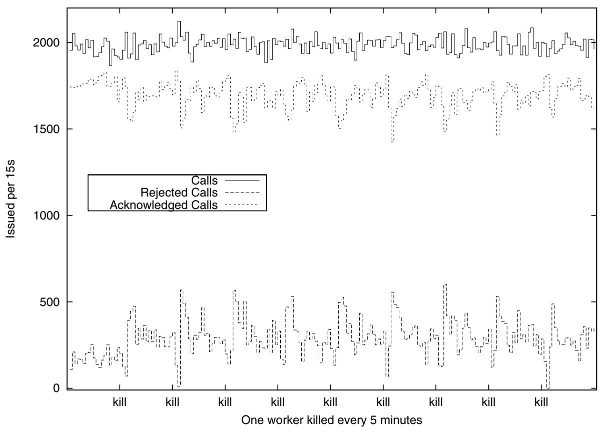


Fig. 2. Repeated failures

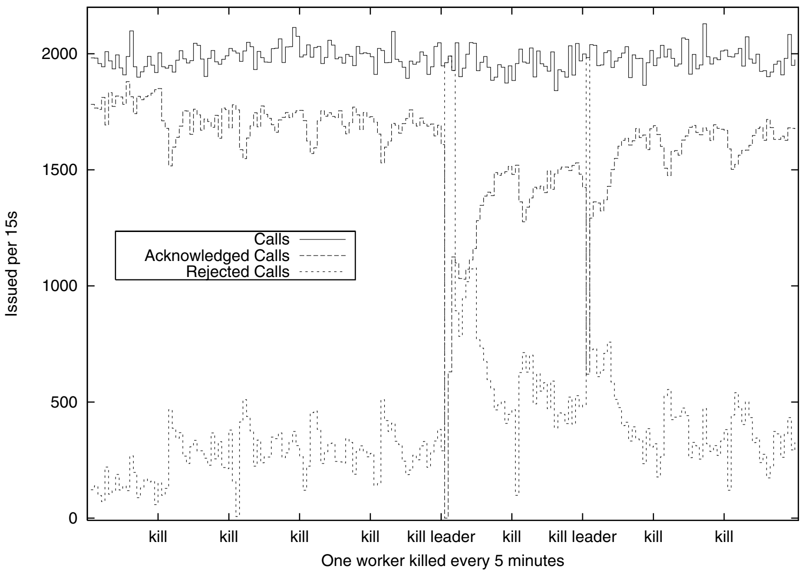


Fig. 3. Several failures

other components are functioning entirely correctly except in the case when one of the other components cease to function properly.

To test the system for single failures, repeated failures and multiple failures we have subjected the system to the following tests:

- One worker node in the system is repeatedly crashed (Figure 2);
- Several different nodes are crashed, one after the other, making sure that the node elected leader is among the nodes crashed as well as the node elected to replace the first leader (Figure 3);

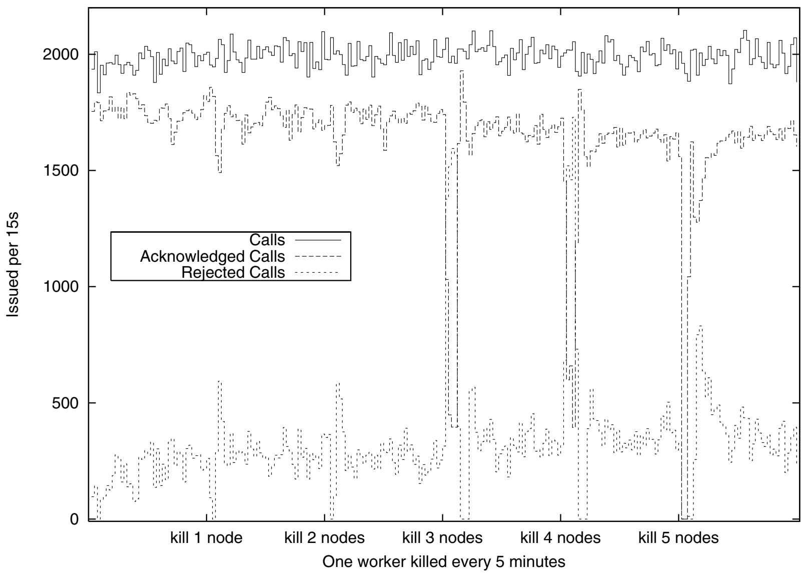
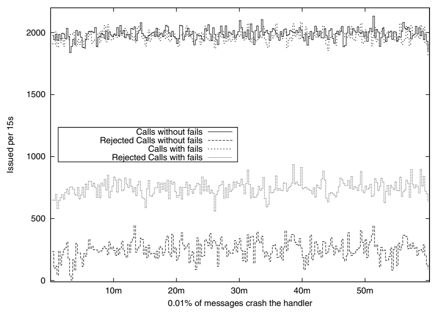


Fig. 4. Multiple failures



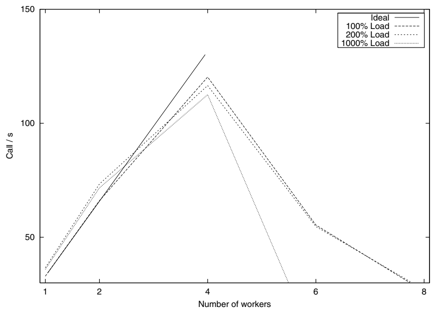


Fig. 5. Service instance failures and Increase requests handled

- Groups of nodes of increasing size are crashed (Figure 4);
- Repeated and multiple failures of the service instance processes (Figure 5).

The reason we have chosen to crash nodes as a means of inducing failures is that the various processes comprising the worker node framework are setup to terminate should any one of the main processes fail; the remaining cases are simulated by individual service instance processes failing.

As the graphs in Figure 2 to Figure 5 shows the system recuperates and regains it former capacity within a reasonable time even in the case where all but one of the nodes has failed.

4.2 Redundancy

Reliable scalable systems are often constructed by composing units with internal replication. For example in the AXD 301 uses pairwise replication with up to 16 pairs of processors [7]. Where some systems replicate hard and software wholesale, in Erlang only the parts of the state required to recover from a failure is maintained in the Mnesia distributed database [11]. While increasing the degree of replication makes the system more reliable (i.e. more nodes can fail without loss of data), the cost of maintaining the replication is quadratic in the degree of replication.

We investigate highest degree of redundancy that can be provided in the DCC without impairing performance. This is investigated by comparing the number of service calls handled per time unit at 100% load with a varying number of fully-replicated worker nodes. For DCC we define 100% load as the number of subscribers that are handled with fewer than 1% of the requests rejected and fewer than 0.1% of message traffic undelivered. Recall that subscribers stochastically generate calls and hand-offs, and hence the relative measure of load.

The service load is provided by traffic generators simulating service calls from subscribers of the services, where the generator generates the number of subscribers per processing element times the number of processing elements.

The result of the experiment is shown in Figure 5, where Figure 5 shows the increase in calls per second on 1 to 8 nodes. The curves plotted are as follows.

X%Load. The number of calls handled by the service at X% load on n nodes with n-times replication.
Ideal. The results we would get if we had linear speedup with increasing workers.

These results show a reasonable speedup of 3.7 times with 4 worker nodes, and that the cost of replication is quite acceptable up to four time replication. Other experiments show that without any replication the system scales well, giving a speedup of 13.95 on 16 worker nodes.

This shows that we can, without impairing the performance, have fourfold redundancy and that we can build systems of larger number of nodes with them grouped in redundancy foursomes.

4.3 Resilience

Overloading the system should result in a graceful degradation of performance. In the context of the DCC an overload is more simultaneous service calls than the system can handle. Like most telecoms systems the DCC controls load by declining requests at the interface.

Resilience is investigated by subjecting configurations of the system to 100%, 200% and 1000% load with a varying number of worker nodes. This experiment uses the same notion of load and traffic generator as in the scalability experiment.

Figure 5 shows several interesting features. Throughput at 1000% is always less than throughput at 200%. Surprisingly, up to 3 nodes there is a small increase in both request and message throughput for both 200% and 1000% load. Thereafter both 200% and 100% result in lower throughput. The reason that an overload at a small number of nodes gives a small increase in throughput is that at 100% load there are times when one or more of the worker nodes execute less than the maximum of service instances.

We have also exposed a one worker system to 50'000% load and the throughput was still 110%. Limitations of the load generation technology prevent measurements of larger systems at such exceptionally high loads.

4.4 Telecoms Characteristics

Two important aspects of telecoms systems are timeliness and resource reclamation, where the time constraint exhibited by the DCC fall into the category of soft real time, meaning that although a very small number of calls may fail entirely, however in all unfailing calls the call control and service payload must be delivered on time. In order to ensure that we conform to the time constraints we have set timers in the traffic generators, and we monitor that voice messages are delivered. During all the tests these properties are checked.

In high availability systems the reclamation of resources is vitally important since any leakage of a resource means that sooner or later the system will crash from lack of that particular resource. One important part of resource reclamation is the management of memory allocated for the call, but since Erlang has automatic memory management this is trivial. The other main problem would be the allocation of central system resources like database entries or tables; all such resources are accessed through a common mechanism where the resources allocated to a service instance are noted in the distributed table entry for the service instance and all resources not deallocated when the instance terminate is automatically deallocated. We have tested this by running the system over long periods of time without any increasing usage of resources.

4.5 Dynamic Adaptability

To provide high availability telecoms systems should adapt quickly to changing demands, such as the resources required. It is not only important that computational and other resources can be added quickly, and without major performance costs during the adaption process, but also that the same resources can be removed when they are no longer needed.

System behaviour as computational resources are added and removed is evaluated as follows. The system is started with 4 worker nodes and nodes are removed until only the leader node remains and then the nodes are added back again until all nodes have been added. The system has been exposed to a 100% load for 4 worker nodes, using the same notion of load as in the scalability experiment.

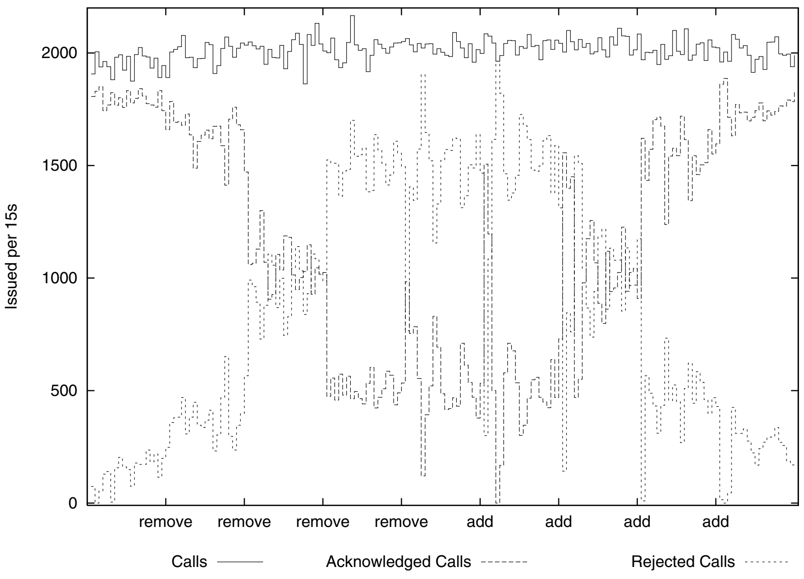


Fig. 6. Dynamic adaptability

The Figure 6 shows the result of the experiment and illustrates the following properties.

- The first half of the graph shows near linear decrease in performance when a node is removed. For example, the approximate number of acknowledged calls per 15s falls from 1800 on 4 nodes to 950 on 2 nodes and 500 on 1 node.
- The second half of the graph shows near linear increase in performance when a node is added. For example the approximate number of acknowledged calls per 15s rises from 500 on 1 node to 950 on 2 nodes and 1800 on 4 nodes.
- The cost of adding/removing a node is small: any fall in throughput as nodes are added and deleted is small.

5 Evaluating the Erlang Fault Tolerance Model

This section critiques the aspects of Erlang fault tolerance model used in the DCC. In implementing the prototype application we have relied on four aspects of Erlang to achieve a fault tolerant system:

- Process encapsulation;
- the ability to monitor other processes and Erlang-nodes for failures;
- OTP library implementations of common supervising design patterns;
- and the OTP distributed database Mnesia implemented in Erlang.

Erlang is a language with true light-weight processes that enables the programmer to use a separate process for each of the tasks the programme has to perform, and yet they are most similar to operating system processes with a non-shared state and independent runtime system managed scheduling. The processes communicate by asynchronous message passing, where the order of messages is only guaranteed for messages from the same sender. This ensures that the process act as a proper encapsulation of computation and a process can only be influenced by another process if it explicitly allows it by receiving messages from that process.

Encapsulation is vital to achieve fault tolerant software, since for the system to be able to handle a faulty component (process) may not unrestrictedly influence other components (processes), otherwise the faulty component may compromise the very components that is supposed to deal with the fault. To encapsulate the faulty behaviour it is also important to minimise the effects of a fault, since if a process controls how it may be influenced it makes it that much easier to detect malign influences.

In order for processes to be able to deal properly with failures in other processes they have to be able to not only determine when such a failure has occurred, but also why, the monitoring facilities of Erlang allows for this information. The monitoring in Erlang is achieved via a bi-directional link that is explicitly set up between processes. When a process fails through an uncaught exception the fact that it has failed and the exception is propagated to all the linked processes. A linked process can choose if it will perform the default action upon receiving the information that a linked process has failed, which is to fail itself, or the process can receive the information as a special message in the message queue.

The notion of links allows the Erlang programmer to properly separate the two concerns of normal execution in his programme and the handling of errors, where one process deals with each aspect. When a programme consists of several tasks each dealt with by one or more processes one comes frequently to the situation where groups of processes with dependencies have to be monitored. A convenient way of organising such groups of process is into supervision hierarchies with several layers of supervising processes being able to deal not only with the failures of individual processes but also with their interdependencies. This common design pattern has support in a library called the supervisor [1].

When dealing with a failure in a component it is often necessary to have a record of the vital part of the failed components state. The state may in the actual implementation be spread over several cooperating processes; this is called a persistent storage since it has to persist the failure of the component. There are many ways of supplying persist storage, e.g., saving logs of state changes to files. In Erlang one common way is to store such information in the distributed database Mnesia that comes with the system. Mnesia is a table-based distributed database implemented entirely in Erlang [11].

5.1 Concise and Clear

The fact that we have true operating system style processes affects the size and clarity of the code in two ways. Firstly, when dealing with systems that have a large natural concurrency like telecoms it is much simpler to implement the concurrent parts as separate processes. The support for processes in Erlang means that very little explicit management of these processes is required. The code size and more importantly the clarity of the code is much influenced by the fact that Erlang has automatic memory management; memory management is normally a great source of complication and errors.

Secondly, the separation between normal execution and fault handling means that the normal execution will not be cluttered up with code dealing with abnormal and erroneous situations and likewise for the fault handling part of the code. This avoids the burden of defensive code where one has to perform torturous and involved testing of every return from a function that may fail and deal with every possible different failure at that point. The failure testing and handling code spread all over the programme when programming defensively is often very similar, and often results in the dangerous habit of cut-and-paste programming. When the common parts of fault handling are separated into a fault handling process one can do without the replication and the result is much smaller and indeed simpler.

Table 1. Erlang and C++ DCC Code Sizes

Erlang					C++			
Part	LoC	Percentage	Modules		Part	LoC	Percentage	Classes
Total	4882	100%	38		Total	21423	100%	
Platform	2994	61%	26		C++	21340	99.6%	81
Testing	1741	36%	11		IDL	83	0.4%	15
Service	147	3%	1					

The size of the Erlang and C++/CORBA DCC are reported in Table 5.1. The number is divided into Platform: which are the parts that are generic and not specific to the DCC service: Testing: which are the parts that are solely involved in the testing and statistics gathering; Service: which are the parts specific to the DCC. The earlier C++ prototype consisted of 21KLOC of C++ and 116LOC of IDL.

5.2 Low Cost Reliability

We argue that the fault tolerance of Erlang is inexpensive in two ways. Firstly the low cost of processes in Erlang means that the cost in resources of having additional processes monitoring fault handling is very low. This can be compared to the cost of performing pervasive testing of all return values or encapsulation performed with proper operating system processes.

The second way in which the Erlang fault tolerance model is inexpensive is in the design and realisation of the system, since in order to achieve a fault tolerant system one has to take into consideration the possible fault and how they can be trapped and dealt with in the design. Being able to separate the issue of fault tolerance gives a simpler design and simplifies the realisation of the design, we do not have to worry about all the point in the system an error may occur, but what conceptually different errors may occur and how we may best deal with them [1].

5.3 Parameterizable and Generic

One very expensive part of fault tolerance in distributed systems is the need to maintain a consistent persistent storage to deal with failures in components on other physical units. It is vital to keep the information needed to recover from a distributed failure to a minimum, although it is always going to be an exponential growth in communication with respect to the number of copies you have of the information. The usage of Mnesia makes it easier to customise the trade off between the number of copies of the distributed data using fragmented database tables. Mnesia allows explicit control of how many copies, in memory or on disk, each table fragment should have. It is thus possible to make it a configuration parameter how many copies should exist, and this parameter may be dynamically changed during the execution of the system.

The use of a high-level language gives the usual benefit that much of the code is actually application neutral and can be reused in other applications, this is even true of the fault tolerance mechanism we use. The most obvious proof, that much of the fault tolerance code is application neutral, is the use of the supervisor library module that is itself implemented in Erlang.

6 Discussion

Summary. This paper reports the investigation into the effectiveness of Erlang for constructing robust telecoms software. We have constructed and measured a typical telecoms application, the DCC. The Erlang implementation meets the functional requirements of the DCC: 99.9% of all voice messages are delivered on time; resources are automatically reclaimed as memory and processes are automatically managed.

Availability experiments show that recovery is fast and the implementation can handle single, repeated and multiple failures of various system components without significant reduction in post-recovery throughput. Redundancy experiments show that the system can provide fourfold redundancy without significant performance penalties.

Resilience experiments show that performance does not significantly degrade under 200% over 1000% load. This would ensure that the system would survive massive overloads, for example from a denial of service attack. Dynamic adaptability experiments show that performance degrades linearly as processors are removed from the system and likewise upgrades linearly as processors are added.

Conclusions. To draw general conclusions about the suitability of Erlang for constructing robust software we have critiqued its fault tolerance model. We argue that it is well-suited to the development of robust telecoms software as it is relatively low cost, parameterisable and generic.

Our work gives some evidence of the conciseness of Erlang: the Erlang DCC is less than a quarter of the size of a similar, but not identical, C++/CORBA implementation. Conciseness reduces both development and maintenance time and costs. We conclude that Erlang and associated OTP libraries are suitable for the rapid development of maintainable and highly reliable distributed products.

Future Work. Following on these initial experiments the teams at Motorola Basingstoke Labs and Heriot-Watt will re-engineer part of a product. We are currently discussing the choice of application with a development team within Motorola.

We also aim to assess the relative merits of other languages for distributed telecoms products by comparing the current DCC implementation in Erlang with the existing in C++/CORBA and JAVA/CORBA and an implementation in Glasgow Distributed Haskell [12] that we have developed.

References

1. J.Armstrong: Making reliable distributed systems in the presence of software errors. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm , Sweden (2003)
2. Fredlund, L., Gurov, D., Noll, T., Dam, M., Arts, T., Chugunov, G.: A verification tool for erlang. International Journal on Software Tools for Technology Transfer **4** (2003) 405–420
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG. 2nd edn. Prentice Hall (1996)
4. Torstendahl, S.: Open Telecom Platform. Ericsson Review (1997)
5. Granbohm, H., Wiklund, J.: GPRS - General Packet Radio Service. Ericsson Review (1999)
6. Hinde, S.: Use of ERLANG/OTP as a Service Creation Tool for IN Services. In: Proceedings of the 6th International ERLANG/OTP Users Conference (EUC'00), Ericsson Utvecklings AB (2000)
7. Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G.: AXD 301: A new generation ATM switching system. Computer Networks **31** (1999) 559–582
8. Wiger, U.: Industrial-Strength Functional programming: Experiences with the Ericsson AXD301 Project. In: IFL'00, Aachen (2000) Presentation Only.
9. Lillie, R.: Implementing dynamic scalability in a distributed processing environment. Technical report, Motorola Labs, Shaumburg, Illinois (1999)
10. Rittle, L.: Distributed Dispatch Architecture Project (Functional Requirements) (1998)
11. Mattsson, H., Nilsson, H., Wikstrom, C.: Mnesia - A distributed robust DBMS for telecommunications applications. In: First International Workshop on Practical Aspects of Declarative Languages (PADL'99). (1999) 152–163
12. Pointon, R., Trinder, P., Loidl, H.W.: The Design and Implementation of Glasgow distributed Haskell. In: IFL'00. LNCS 2011, Aachen, Germany (2000) 101–116

Functional Apportioning of Safety Requirements on Railway Signalling Systems

Ola Løkberg and Øystein Skogstad

SINTEF ICT, Information Security and Safety, NO-7465 Trondheim, Norway
`{ola.lokberg, oystein.skogstad} @sintef.no`

Abstract. A method for apportioning of Tolerable Hazard Rates (THR) on railway signalling equipment through a defined set of related safety critical functions is presented. For this approach to be effective, a number of steps have to be taken, involving political, economical as well as technical considerations: How many casualties pr. year (TLL – Tolerable Loss of Life) due to railway operations shall be accepted by the society? How many of these casualties shall be allowed attributed to the signalling systems? How can this signalling quota be apportioned onto a set of safety critical functions? How can the safety requirements of these functions be further apportioned onto the physical equipment realizing the functions, eventually making it possible to specify and validate the actual equipment being installed: What is the expected Hazard Rate (HR) of the defined safety critical functions and what are the consequences if they fail, i.e. if a hazard occurs?

The underlying study of this paper has been carried out as part of a contract with the Norwegian railway authority Jernbaneverket.

1 Introduction

Currently, there is a shift to imposing safety requirements on railway *functions* rather than railway *equipment*. This view is advocated by the European Standards EN50126 [5], EN50128 [6] and EN50129 [7] (the railway sector implementation of the IEC 61508). A functional decomposition approach has also been adopted by the SAMNET/SAMRAIL consortium preparing the ground for the European Railway Agency (ERA) and the specification of Common Safety Targets (CST) in the new European Railway Safety Directive (2004/49/EC) [8].

2 European Railway Safety Standardization

In Europe at the beginning of year 2005, a lot of railway safety standardization activities are taking place, or are about to start. The main objectives of these activities include:

- Implementation of a new Railway Safety Directive [8], including development of the safety indicators, methods and targets required to have a common way of specifying and measuring safety.
- Get the European Railway Agency up and running (see Sect. 0)

2.1 Railway Safety Directive (2004/49/EC)

Requirements on safety of the different subsystems of the trans-European rail networks are laid down in the directives 96/48/EC (high-speed), 2001/16/EC (conventional) and 2004/50/EC (amendments to the previous two). However, those directives do not define common requirements at system level and do not deal in detail with the regulation, management and supervision of safety.

Directive 2004/49/EC [8], hereafter called the Railway Safety Directive, was established as an acknowledgement to the need of satisfying these requirements by establishing a common regulatory framework for railway safety within the European Union. Of particular interest in this context was to harmonise the development and management of safety, including the development of Common Safety Targets, CSTs.

Common safety targets (CSTs) shall define the minimum safety levels that must be reached by the different parts of the railway system and by the system as a whole in each Member State, expressed in risk acceptance criteria for:

- individual risks relating to passengers, staff (including the staff of contractors), level crossing users, unauthorised persons on railway premises
- societal risks

2.2 European Railway Agency (ERA)

The European Railway Agency ERA (hereafter called the Agency) was proposed by the European Commission in connection with the "Second Railway package" on January 23. 2002 and later entered into force on May 1. 2004. The Agency has been established to provide the Member States and the Commission with technical assistance within the fields of railway safety and interoperability.

The Agency will provide the technical assistance necessary to implement Directive 2004/49/EC, the Railway Safety Directive. This implies that the Agency shall:

- prepare and propose Common Safety Methods (CSM) and Common Safety Targets (CST)
- finalise the definition of the Common Safety Indicators, CSI, and perform continuous monitoring of safety performance through these indicators

2.3 SAMNET/SAMRAIL

The SAMNET and SAMRAIL projects are both financed under the "Competitive and Sustainable Growth" area of the 5th Frame Programme. The two projects are closely linked: The partners in the SAMRAIL project are also members of SAMNET, and the managerial and technical activities of the two projects are coordinated. The tasks of SAMNET/SAMRAIL are closely related to the tasks of the European Railway Agency, the work performed within projects is also to be continued by the agency.

The most interesting aspect of the SAMNET/SAMRAIL work in this context is the work on functional apportionment of safety requirements. This work is based on a functional decomposition scheme proposed by AEIF, the European Association for Railway Interoperability. This scheme divides (total) railway operation into a total of 12 [9] first level functions. Function "Operate a train" is applicable to the railway signalling system and is the umbrella under which the signalling system safety critical functions SCF_1 and SCF_2 presented in Sect. 0 can be located.

3 CENELEC Norms

The CENELEC norms EN 50126 [5] and EN 50129 [7] contain nomenclature and recommendations of processes with respect to safety and risk analysis and the compilation of accept criteria. In addition, the CENELEC norm EN 50128 [6] deals with software included in safety critical applications.

The purpose of this paper is to present a method of assigning safety requirements in the form of THR (Tolerable Hazard Rates) to functions. A function in this context encompasses the required collection of hardware, software and operational procedures to implement the specified functionality. Software is not distinguished as a separate unit, the system perspective taken by EN 50129 is therefore sufficient in this context.

3.1 Railway Authority vs. Supplier

The distribution of responsibility between the Railway Authority[1] on one hand and the Supplier of railway equipment on the other hand is illustrated by the following figure from EN 50129 Annex A:

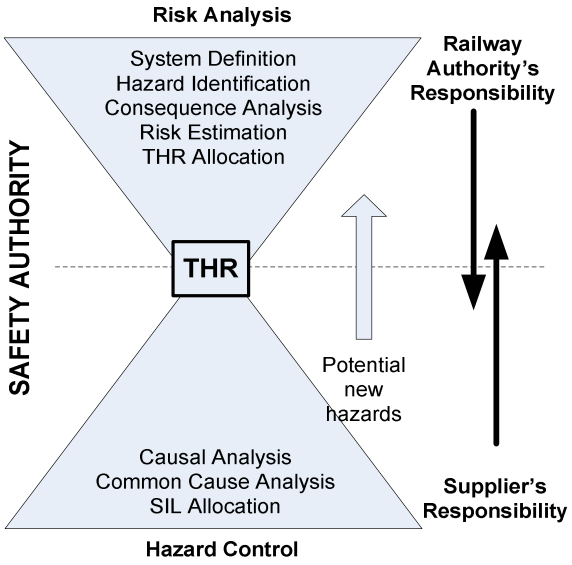


Fig. 1. Global process overview (from EN 50129 [7], Annex A)

[1] A Railway Authority may in general be an (infrastructure) owner of one or more parts of the railway system, an operator, a maintainer of one or more parts of the railway system.

Based on this figure the essence of EN 50129 with respect to determination of THR and SIL requirements can be summarized as follows:

Responsibility of Railway Authority:

- Define a set of safety critical functions for the signalling system.
- For each defined safety critical function: Define a set of safety integrity requirements.
  The definition of safety integrity requirements is a systematic process including the identification of hazards, consequence analysis and risk estimation.
- The result of this process is a set of Tolerable Hazard Rates for the set of defined safety critical functions.

Responsibility of Supplier:

- Define the system architecture as well as allocate system functions to the different parts of this architecture to meet the safety requirements.
- Apportioning function related THR (received from the Railway Authority) to the subsystems/components required to implement the corresponding functions.
- Determine corresponding SIL classes from the apportioned THR values.

4 Safety Critical Functions

4.1 Why Safety Critical Functions

Currently, THR safety requirements to signalling systems are often imposed on signalling systems as a whole. This makes it more difficult for large systems to fulfil the requirements, simply because the determined THR value (being equal to all signalling systems irrespective of size) must be apportioned on more equipment in a large system. Every component in a large system must then individually be safer than if it has been used in a smaller system.

Basically, it is the safety level of functions which is important for the Railway Authority, not how these functions are realized. When imposing safety requirements on functions (in contrast to equipment), the safety requirements better reflect what's important in addition to that they are more robust with respect to changes in technology.

According to EN 50129, safety requirements in railway signalling systems shall be set to functions. The actual definition of these functions, and on which level they shall be defined, is however not determined by the standard.

4.2 Top Events

There is a close connection between safety critical functions on one hand and top events (accident categories) on the other: The total set of safety critical functions shall cover all relevant top events.

Related to the signalling system this means that the defined set of safety critical functions must cover all top events caused by faults in the signalling system.

The following set of top events (accidents) can be caused by faults in the signalling system:

- Collision train–train (head to head, head to tail, head to flank)
- Derailment

Collision train-object accidents are usually not concerned with faults in the signalling system and are therefore not considered in our context.

4.3 Defined Safety Critical Functions

According to the selected top events presented in Sect. 0, a set of safety critical functions covering both train – train collisions and derailments must be determined.

Basically, a signalling system has the following main function: Ensure that a train can move from A to B in a way that accidents are avoided. For this main function to be realized, two corresponding sub functions must operate correctly:

1. All signal aspects being part of the route from A to B are set up correctly so that the train driver receives the correct and required information.
2. All points being part of the route from A to B are set up correctly and stay in the correct position until the train has passed by.

The automatic train protection aspects (ATP, ATC) defending the train from passing signals at danger and from over speed are not considered in this paper.

Correspondingly, the following *types of* signalling system safety critical functions have been defined:

– **SCF$_1$**: The function of providing signal aspects according to the current conditions for train movement.
– **SCF$_2$**: The function of maintaining correct position of points within the time interval when a faulty position can not be signalled to the driver in time.

While the wording of SCF$_1$ is fairly straight-forward, an explanatory comment should be made concerning SCF$_2$: The mechanisms for detecting the actual physical rail position within a point is an integral part of the signalling system and interlocked with the signal. This means that if a point for some reason does not obtain the correct physical position, this will be reflected in the signal related to that point (SCF$_1$). If, however, the train is within braking distance of the point or is actually passing the point, the information about a changed position within a point is received too late to be of any use, SCF$_2$ has therefore been defined to cover this situation without overlapping SCF$_1$.

To verify that the apportionment of THR to the defined types of safety critical functions SCF$_x$ is appropriate irrespective of technology, operational procedures etc., it is important to have suitable monitoring- and logging mechanisms making it possible to connect fault and accident situations to the appropriate SCF$_x$.

5 From TLL to THR for SCF

One of the goals of the study behind this paper was to examine how equipment related THR requirements can be in accordance with national objectives for accident risks related to all railway activity in Norway. This objective is often designated as PLL, *Potential Loss of Life*, indicating the maximum number of casualties caused by railway activity per year which can be accepted. However, to be more in harmony with the corresponding designation THR (*Tolerable Hazard Rate*), we will instead use the designation TLL (*Tolerable Loss of Life*). In addition to being symmetric to THR, the designation "TLL" also has the property that it through the word "tolerable" clearly indicates that it is not a computed, but a "politically determined" value.

5.1 THR for Safety Critical Functions

Assuming that the national safety objective is expressed as a politically determined maximum casualty value, $TLL_{National}$, the question is:

How can this objective be transferred into safety requirements for the corresponding set of Safety Critical Functions, SCF_1 and SCF_2?

The process of determining safety requirements for SCF_1 and SCF_2 for the signalling systems from the $TLL_{National}$ value goes through a number of steps. These are:

1. Determine the effect of signalling system failures
2. Determine the amount of hazards caused by the signalling systems
3. Determine the probability of an accident to occur given a hazard in the signalling system
4. For each specific signalling system application: Apportion the safety requirements on the individual *instances* of SCF_1 and SCF_2.

As we are lacking data to proceed systematically along this way, we have to use "bold questimates" to establish a first set of overall THR values. The reasoning behind this estimation is explained in the sequel.

The Effect of Signalling System Failures. There is a comprehensive statistical material for Norwegian railway accidents. However, none of these accidents are with certainty caused by the signalling system. There is therefore no available statistics known to the authors showing any contribution to top events from signalling system failures.

Consequences of accidents can be grouped into *consequence classes*, distinguished by the number of casualties caused by the accident. The risk analysis performed in [1] shows that for many of the analyzed scenarios, failures in SCF may be catastrophic (more than 10 casualties, highest consequence class) given that a fault has occurred. Using the ALARP (As Low As Reasonably Practical)-principle on the risk matrix presented in [2] gives an acceptable maximum number of 0.1 signalling system failures in the highest consequence class for the Norwegian railway per year.

Hazards Caused by the Signalling Systems. All railway authorities maintain a comprehensive statistical material for failures related to the railway infrastructure, including a classification of whether these failures were safety critical or not. However, this large amount of data has usually not been analysed with respect to creating a statistic for *safety critical failures*. From the available material it is therefore some way to go to determine how many of the total number of failures were hazards.

As discussed in the previous paragraph, available statistics contain no accidents which by a high probability can be attributed to the signalling system. Reference [4] suggests that signalling systems could be assumed to cause 1% of total railway system hazards. In the lack of available statistics we will use this figure. As already mentioned, the risk analysis [1] shows that SCF failures may be catastrophic. Conservatively it is therefore correct to attribute all signalling system failures to the highest consequence class (more than 10 casualties).

Probability for an Accident to Occur. The risk analysis presented in [1] shows that faults in a safety critical function SCF will with a high degree of probability cause an accident. An erroneous (too) liberal signal aspect will most often occur in situations where trains are present. The conservative estimate for the probability of an accident to occur given a hazard of the signalling system is therefore 1. Our first estimate of this probability is 10%.

Altogether, the reasoning in this section leads to a resulting THR for the Norwegian railway signalling system of 0.01 highest consequence failures per year. This is shown in the Table 2.

It must be emphasized that the figures in the table are *estimates*, derived from the discussions in this and the preceding paragraphs.

Table 1. THR for the Norwegian railway signalling systems

Factor	Value	Corresponding THR [failures/year]
Maximum rate for accidents of highest consequence class		0.1
Hazards due to the signalling systems	1%	0.001
Probability for an accident to occur given fault in SCF	10%	0.01

THR Apportioned on Total Set of SCFs. We have previously assumed that the defined set of SCFs cover all safety critical functionality of a signalling system. Consequently, the THR requirement of 0.01 failures per year applies to all implementation instances of the defined set of SCFs in the Norwegian railway infrastructure (i.e. all signalling systems).

This total THR should not be equally apportioned on SCF_1 and SCF_2. The risk analysis presented in [1] shows that there are different accident scenarios for the two SCFs: When failing, SCF_1 is likely to have more severe consequences than SCF_2. We therefore set THR for SCF_1 equal to the total THR value above: 0.01 failures per year. As far as SCF_2 is concerned, less severe consequence when failing implies that the THR requirement to this safety critical function can be weakened. As an initial estimate, two decades down, i.e. 1 failure per year are regarded a suitable value here.

To summarize, the following THR requirements should be set to (all instances of) the two defined safety critical function of the signalling systems of Norway:

SCF_1 (all instances) : THR = 0.01 failure per year
SCF_2 (all instances) : THR = 1 failure per year

THR Apportioned on Individual Instances of SCF. By estimating the total amount of signals and points contained in the Norwegian railway signalling systems as well as their number of possible states, a total of 10000 instances of SCF_1 and 5000 instances of SCF_2 have been estimated. This gives the following individual THR values:

SCF_1 (per instance) : THR = 10^{-6} failure per year $\approx 10^{-10}$ $[h^{-1}]$
SCF_2 (per instance) : THR = $2*10^{-4}$ failure per year $\approx 2*10^{-8}$ $[h^{-1}]$

It must be emphasized that the calculated requirements presented in this section are based on ***estimates****. However, we believe that the precision is good enough to claim that the requirements are anchored in the national safety objectives, expressed as a maximum number of fatalities per year caused by total railway operation. We also believe that the calculations in a clear and understandable way show how large a portion of the total safety requirements should be attributed to the signalling systems.*

Comparison with Today's Equipment Based Safety Requirements. Reference [3] contains today's safety requirements for Norwegian railway signalling systems. Although the requirements included in [3] are equipment based and expressed for a complete signalling system, they are in the same order of magnitude as the function-based set of requirements presented in this paper. From this we may draw the conclusions that the THR requirements presented in this paper both may be realized with available technology as well as does not represent a degradation of today's safety level of signalling systems. An actual case study is however required to verify this hypothesises.

5.2 THR's Relations to Other Railway System Parameters

Basically, trains can be guided or controlled in two ways: *Automatically*, when the signalling system is operational, or *manually*, when the signalling system is down. The two safety critical functions SCF_1 and SCF_2 defined in Sect. 0 both apply to automatic train control. Further, THR requirements are always related to random faults. However, statistical data show that systematic faults contribute significantly to accidents and hazard. Consequently, system safety can not be characterized by THR requirements alone without regarding other system parameters.

System Availability. A system's availability is a measure of whether it can be expected to be operational. [5] defines the term "Availability" as *the ability of a product to be in a state to perform a required function under the given conditions at a given instant of time or over a given time interval assuming that the required external resources are provided.*

Note that a system's availability is not dealing with safety: A system can be unavailable due to a variety of reasons, not only because the system's safety functions are not operational. When a railway signalling system is unavailable, all signal aspects are set to red ("stop") which is considered a safe state as seen from the signalling system.

Some sources (e.g. ref. [4]), claim that manual and automatic train control (through the signalling system) contribute equally to the total risk of railway operation. If that is the case, and the availability value for signalling systems is 99.5 % (ref. [4]), the hazard rate during periods of manual train control is 200 times larger than the corresponding rate with automatic control. This illustrates that there is obviously a trade-off whether to use resources to improve THR values of the signalling systems, effective during normal automatic train control, or to improve the system's availability to reduce the risks imposed through manual control.

Random vs. Systematic Faults. THR reflects the expected rate of hazards due to safety related, random equipment faults. Such faults are random because they are triggered by random events as a malfunctioning component.

Systematic faults are faults which will re-occur if the situation producing the fault is recreated. Examples of systematic faults are faults in specification, design, software coding as well as operational procedures. Generally for the PES (Programmable Electronic System) type of technology, systematic faults are claimed to occur as often as random faults.

However, it is of minor importance to the public whether a number of people have been killed in a railway accident due to a random or a systematic fault. Taken together, signalling system THR values may therefore be applicable to only one forth of the total risk (random faults occurring during automatic train control). Even so, we choose not to change the THR values for the signalling systems since the correction would be minor compared to the decadic approximations used to arrive at these figures.

Size of Installation. In this paper, THR is apportioned on safety critical functions, SCFs. A large installation will have a larger number of SCFs than a smaller installation. Because the number of SCFs is directly related to the number of signals and points, it can be argued that the number of SCFs is directly proportional to the size of the installation. It will thus scale very well with installation size.

Therefore, with respect to safety requirements, the same equipment can be used both for large and small installations. This makes it easier for the railway authority to standardize on specific types of equipment, in turn reducing costs with respect to maintenance, spare parts storage and personnel training.

6 From THR for SCF to THR for Equipment

We have in Sect. 0 shown how a TLL (Tolerable Loss of Life) national railway safety requirement value can be distributed onto a set of safety critical functions, SCF.

The next step is to distribute this further onto the collection of equipment required to implement the SCFs. The method for this is shown in the following figure:

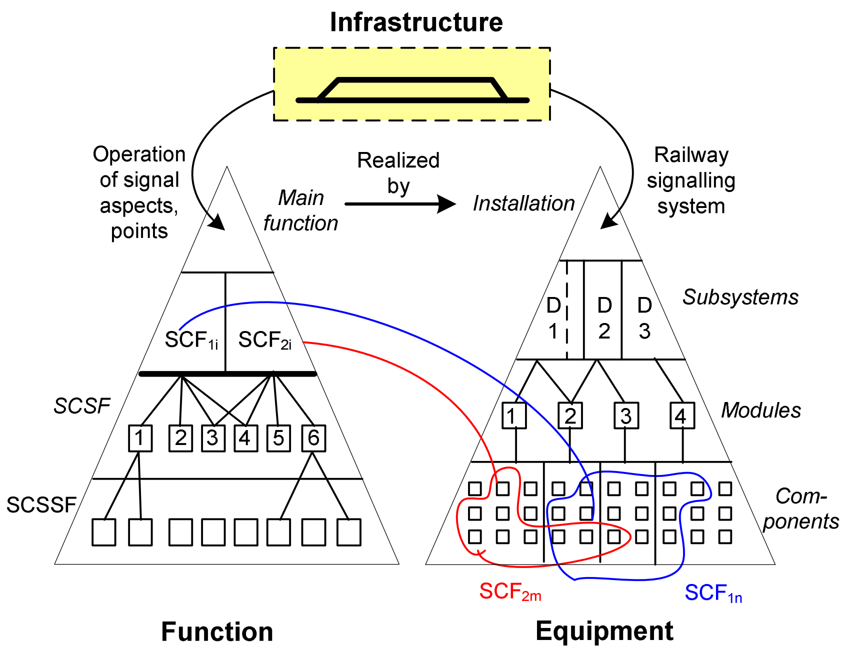


Fig. 2. Function vs. implementation

6.1 Duality Between Function and Equipment

Generally speaking, any technical construction or mechanism can be represented by a dual view: Either by the *functions* it performs, or by the *equipment* by which it is realized. For a system, both function and equipment are hierarchical in nature and can thus be illustrated as a pyramid as shown in Figure 2.

Applied to the topic of this paper, a railway **Infrastructure** can thus be viewed as a duality between its **Main Function** (**Operation of signal aspects, points**) and the **Installation** (**Railway signalling system**) realizing this function.

The concrete structure of these pyramids and the mapping between them depend on the actual system solutions being selected. Without such knowledge, only the relationships/proportions between the different THR/HR values can be given, not their absolute values.

Function Pyramid. As described in Sect. 0 the safety critical functionality Operation of signal aspects, points will be represented by two safety critical functions, SCF_1 and SCF_2. *All safety critical functionality within the scope of an interlocking system is covered by these two safety critical functions.*

Further, within an infrastructure there will generally be several instances of an SCF. These instances are not necessarily realized identically, this will often depend on where in the infrastructure they are located. The method therefore supports that the infrastructure contains N instances of SCF_1 (denoted SCF_{1i}, i = 1...N) and M instances of SCF_2 (denoted SCF_{2i}, i = 1...M).

Due to the effect of any barriers and/or safety redundancy, the resulting Hazard Rate HR of any given instance of an SCF (denoted SCF_{xi}) will always be less than or equal to the sum of Hazard Rates of the equipment required to realize the instance:

$$HR_{SCFxi} \leq \sum HR_{Equipment} \quad \text{(all eq. realizing } SCF_{xi}\text{)} \tag{1}$$

Any instance SCF_{xi} of an SCF must be apportioned a corresponding safety requirement in form of a *Tolerable* Hazard Rate value, THR_{SCFxi}. For the safety requirement to be fulfilled, this THR value must be greater or equal to the corresponding (calculated) HR value, i.e.:

$$THR_{SCFxi} \geq HR_{SCFxi} \tag{2}$$

The defined types of safety critical functions SCF_1 and SCF_2 may use the same equipment parts and are thus not statistically independent. In addition, the same equipment may be a part of the realization of several instances of the same type of SCF (SCF_x), typically this will be the case for the central interlocking system. Consequently, the sum of HR for all instances of SCF_1 and SCF_2 in the installation will be higher than the HR of the total installation:

$$HR_{Installation} < \sum HR_{Equipment} \quad \text{(all eq. realizing all instances of } SCF_1 \text{ and } SCF_2\text{)} \tag{3}$$

Safety Critical Sub Functions (SCSF). It may be necessary (e.g. in case the equipment realizing one SCF_x come from several suppliers) to further divide SCF_x into safety critical *sub* functions $SCSF_x$. These can in principle again be divided into new sub functions ($SCSSF_x$) and so forth. In general, the same sub function (e.g. train location detection) may be a part of both SCF_1 and SCF_2.

If it is required to divide SCF_x into sub functions $SCSF_{xy}$, the sum of THR for all $SCSF_{xy}$ will be equal to the THR for the corresponding SCF_x:

$$THR_{SCFx} = \sum THR_{SCSFxy} \quad \text{(for all } SCSF_{xy} \text{ of } SCF_x\text{)} \tag{4}$$

However, because the same sub-function SCSC can be a part of both SCF_1 and SCF_2,. the sum of THRs for all SCF_x will always be less or equal to the sum of THRs for all sub-functions $SCSF_{xy}$:

$$\sum THR_{SCFx} \text{ (in total for all x)} \leq \sum THR_{SCSFxy} \text{ (in total for all x and y)} \tag{5}$$

This division into sub functions SCSF, and if required sub-sub functions SCSSF, may be performed until one and only one supplier supplies equipment to one

SCF/SCSF/SCSSF. When so, there will be a pure function-equipment interface between the Railway Authority and the suppliers.

Equipment Pyramid. The safety critical main function Operation of signal aspects, points is realized by a Railway signalling system. The Railway signalling system is in turn realized by a number of Subsystems, in Figure 2 denoted D1 to D3. The subsystems may be overlapping in the way that several subsystems rely on some common service (e.g. central logic). Subsystems D1 and D2 are in Figure 2 shown as overlapping.

Every subsystem is in turn realized as one or more **Modules**. Because subsystems may be overlapping, one module may belong to several subsystems. Module 2 in Figure 2 is an example of this.

Each module is further realized by one or more **Components**. Each component will belong to one and only one module. Generally speaking, a number of components from different modules (but not necessarily all components within one module) will be a part of the realization of one SCF. Because one component can be a part of the realization of both SCF_1 and SCF_2 as well as due to barrier/redundancy effects, the sum of the hazard rates HR for all components C used to realize SCF_x will always be equal to or larger than HR for SCF_x, that is

$$HR_{SCFx} \leq \sum HR_C, \quad \text{in total for all components C included in the realization of } SCF_x \qquad (6)$$

6.2 Method for Apportioning THR on Equipment

The method for determination and verification of safety requirements based on THR to safety critical functions is based on the duality between the function and equipment pyramids as shown in Figure 2. The method can be described as follows:

1. **Starting point.** THR requirements are apportioned to all instances of SCFs within the actual infrastructure. This can either be done on a generic basis as shown in Sect. 0, or on a installation specific basis facilitating special treatment of the individual installation, taking other factors than the actual number of SCFs into consideration.
2. **Transform SCF THR requirements to THR requirements to equipment.** The actual method is indifferent to whether there are one or several suppliers of equipment realizing the SCF. However, with only one supplier the SCF itself constitutes the interface between the Railway Authority and the supplier. With several suppliers, the SCF and the corresponding THR requirement must be apportioned (by the Railway Authority) into sub functions SCSF before handed over to the supplier.
3. **Verification of selected equipment against THR requirement to SCF.** This is done by computing the resulting HR for the SCFs (see Sect. 0). If the specified THR requirements are not satisfied, the realization of the SCF must be changed, either by using other (and less error prone) components, or by changing the architecture, e.g. by incorporating a higher degree of redundancy.

7 Verification of THR Requirement

Normally it is the task of the supplier to prove that his selection of equipment satisfies the THR requirement for SCF_x. It is generally acceptable that this is done as an analytical computation based on reliability theory. Such a computation must include the following elements:

1. Identify the individual equipment (Figure 2: Subsystems, Modules, Components) required to realize the relevant instance of SCF_x.
2. Determine the failure rate for this equipment from the manufacturer's specifications.
3. Determine the Hazard Rate (HR) for the same equipment, e.g. by carrying out an FMEA (Failure Mode and Effect Analysis). Normally the HR will be significantly lower than the failure rate (not all failure modes will be safety critical, redundancy).
4. The corresponding Hazard Rate for the different realizations of SCF_1 and SCF_2 are computed using the equipment HR as input. The effect of any barriers (e.g. human) must be included.

If the analysis shows that the resulting Hazard Rate for SCF_x is in the same order as the corresponding THR, the requirement should be considered as satisfied. Due to the uncertainty in the input data a sensitivity analysis may be requested if the THR requirement is not completely satisfied. Factors as system availability and the effect of systematic faults (see Sect. 0) should also be taken into consideration if the result is close to the requirement.

8 Conclusion

This paper has presented a method of imposing safety requirements on safety critical functions, SCFs, rather than directly on equipment. By doing this, the requirement of the individual SCF will be independent of installation topography as well as implementation technology. In the longer term, the confidence in railway safety should also be improved by the fact that going through SCFs forces the railway safety management to focus directly on safety critical functions.

It can be argued whether going through safety critical functions rather than directly on equipment makes it more easy or difficult to verify safety requirements. Experience indicates that there is no difference: In both cases, traditional safety and reliability analyses techniques for electronic/electromechanical systems must be applied.

References

1. Løkberg, O., Øien, K., Hokstad, P., Skogstad, Ø.: Utvikling av Tolerable Hazard Rates for signalanlegg (in Norwegian). SINTEF Report STF90 F05005 (2005)
2. Jernbaneverket, Sikkerhetshåndbok (in Norwegian). Document. no. 1B-Sikkerhet (2003)

3. Jernbaneverket, Teknisk regelverk. Signal/prosjektering (in Norwegian). Document. no. JD-550 (2004)
4. Andersen, Terje. Sikkerhetskrav til sikringsanlegg, rev. 01 (in Norwegian). DNV report 2002-0157 (2002)
5. EN 50126. Railway applications – The specification and demonstration of reliability, availability, maintainability and safety (RAMS) (1999)
6. EN 50128. Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems (2001)
7. EN 50129. Railway applications – Communications, signalling and processing systems – Safety related electronic systems for signalling (2003)
8. DIRECTIVE 2004/49/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 29 April 2004 on safety on the Community's railways and amending Council Directive 95/18/EC on the licensing of railway undertakings and Directive 2001/14/EC on the allocation of railway infrastructure capacity and the levying of charges for the use of railway infrastructure and certification (Railway Safety Directive) (2004)
9. Report on the Representative Architecture. Revision 1.8, December 2002. Report issued by the AEIF project team (2002)

Automatic Code Generation for PLC Controllers

Krzysztof Sacha

Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warszawa, Poland
k.sacha@ia.pw.edu.pl

Abstract. The paper describes a formal method for automatic generation of programs for PLC controllers. The method starts from modeling the desired behavior of the system under design by means of a state machine with the ability to measure time and ends-up with a complete program written in a ladder diagram language. The model is formal, yet readable, and can be verified against the behavioral and safety requirements. The conversion of the model into a program is done automatically, which reduces the need for further program verification.

1 Introduction

Computerized control systems are used in many industrial applications in which a malfunction of the system can endanger the environment or human life. The systems that are used in such application areas are expected to exhibit always an acceptable behavior. Such a property, often referred to as dependability, is a system-level attribute, which must be considered at hardware as well as software level. The typical hardware devices used in industrial control are Programmable Logic Controllers (PLC) that are designed in such a way that promotes reliability and predictability of the controller operation, and makes the design of time- and safety-critical systems easier.

PLC is a specialized computer, which has a set of input interfaces to sensors, a set of output interfaces to actuators and an operating system that manages the repeated execution of the following cycle:

- Reading all input sensors and storing the read data as values of input variables.
- Processing the inputs and computing the new values of the output variables.
- Updating all outputs with the current values of output variables.

The maximum duration of each execution cycle is bounded and guaranteed by the operating system. This introduces an explicit granularity of time: An input signal that does not hold for at least the maximum duration of the cycle can remain unnoticed by the PLC. Moreover, a response to an input signal cannot be expected earlier than in the next consecutive cycle of execution.

Programming of a PLC deals with the computing phase of the execution cycle only. The core part of the computation relates to calculations of Boolean conditions that define the current state of the controller and the current values of two-state output variables. The programming languages, standardized in [1], include: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD) and Function Block Diagram (FBD).

A PLC can be used alone, but in many real applications it is a part of a bigger system that consists of several PLCs and computers coupled and working together. The development of such a system can be driven by a set of UML-based models [2] that describe the required behavior of the system as a whole, and of all of its components. The conceptual tool that is offered by UML to model this type of processing that is done by a PLC is state diagram – a model that describes the states an object can have and how events (input signals) affect those states over time.

The goal of this paper is to describe a formal method for automatic programming of PLCs, which uses a subset of UML-based state diagram model to define a correct control algorithm, and to implement the algorithm automatically using the ladder diagram language. The choice of the target language (LD) has been motivated by the widespread use of certified programming environments offered by the majority of PLC manufacturers. Nevertheless, generation of a program in C language executed under a POSIX-type operating system is also possible. The advantages of the method are simplicity that has been verified in student labs, easy interfacing to UML-based software development process and tools, and the possibility of automatic code generation. The verification of the program correctness can be performed at the model level.

The paper is organized as follows. Section 2 provides the reader with a short overview of the subset of UML-based state diagram model that is used in the paper. Section 3 introduces a formal definition of the finite state time machine that defines the semantics of the state diagram model. The process of converting a finite state time machine into a program, written in the ladder diagram language, is described in Section 4. The description is illustrated using a case study of a bottling line controller. Final remarks and plans for future work are given in Conclusions.

2 State Diagram

Basically, state diagram is a graph that shows how an object reacts to events that originate in the outside world. It consists of states that capture distinct modes of the object behavior and transitions between states that are caused by events and accompanied with actions. Relating the model to the structure of PLCs one can note that events correspond to the occurrences of input signals, and actions correspond to changes of the output signals. The modeling concept is simple and consistent with the mathematical theory of finite state machines. UML adds further elements to this model:

- Entry and exit actions of a state that are executed on entering and exiting the state.
- Guards, i.e. Boolean conditions that enable or disable transitions.
- Internal transitions that are handled without causing a change in state.
- Deferred events that are memorized for handling in another state.
- Time events that correspond to the expirations of predefined periods of time.

Entry and exit actions of a state do not add any new semantics to the model as they can easily be reassigned to transitions that input or output the state. Guards deal with the attributes of an object and do not apply to modeling of PLCs. Internal transitions and deferred events violate the rule that the only memory of an object is state, and therefore are excluded of the model that is used in this paper. If an event could make a permanent change to an output or had to be memorized, an explicit change to the object state must be shown.

A substantial extension to the model of a finite state machine is the introduction of time events. Such an event originates inside the modeled object, and breaks the rule that the reaction of the object to an external event depends on the current state only. An additional memory of timers that measure the flow of time is needed. An attempt to describe the extension is given by the theory of timed automata [3]. Still another formal model of a finite state time machine is introduced in Sect. 3.

A disadvantage of a state diagram, as described above, is the lack of tools for managing complexity of real systems that can have hundreds or even thousands of states reflecting very general or very detailed properties of the modeled objects. One way to capture the behavior of such a complex system is to describe its behavior using many levels of abstraction. UML offers hierarchical state diagrams, in which a state can have sub-states, each of which can be shown in another state diagram. A transition that leads to a super-state activates the internal state diagram in its initial state. A transition that roots in a super-state can occur in each of its internal sub-states.

The presence of nested states leads to quite a new concept of a history indicator that is used to memorize the last sub-state on exit of a super-state, so that it's possible to go back to this sub-state at a later time. History indicator adds memory to the model – a discussion of this feature is given in Sect. 4.2.

3 Finite State Time Machine

Finite state machine is a recognized tool for defining the algorithms of processing the enumerative sets of events. The automaton-like graphical models are formal, as well as understandable to engineers and computer programmers. What is missing to a classical finite state machine is the ability to model time. In this section we define a new model of a finite state time machine that adds time to the classical Moore automaton.

Definition. A *finite state time machine* is a tuple $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$, where

S is a finite set of *states*,
Σ is a finite set of *input symbols*,
Ω is a finite set of *output symbols*,
Γ is a finite set of *timer symbols*,
$\tau: \Gamma \to S \times R^+$ is an injective function, called *timer function*,
$\delta: (S \times \Sigma \cup S \times \Sigma \times \Gamma) \to S$ is a function, called *transition function*, which is total on $S \times \Sigma$ and partial on $S \times \Sigma \times \Gamma$: $(s,a,t) \in Dom(\delta) \Leftrightarrow (\exists r \in R^+)[\tau(t) = (s,r)]$
$s_0 \in S$ is the initial state,
$\varepsilon \in R^+$ is the granularity of time,
$\omega: S \to \Omega$ is an output function.

Notation: R^+ is the set of positive real numbers, $Dom(\delta)$ is the domain of function δ. Cardinality of a set X will be denoted $card(X)$.

It can be noted from the above definition that a finite state time machine is finite, and looks much like a Moore automaton with three additional elements: Γ, τ, ε. The rationale that stands behind the timer symbols can be explained as follows. The only memory of a Moore automaton is state. Adding time to such an automaton adds an additional kind of memory that stores durations of time intervals. This additional kind of memory is explicitly shown as a set of timer symbols. Each timer symbol will be converted in the implementation process into a timer device that measures time.

For an example, consider a train-detecting sensor [4] that signals '*a*' if a train is approaching, '*b*' if not, and '*Error*' if a failure of the device has been detected. The sensor can stutter for a time Δt after a train has passed the sensor. The control system is expected to filter the stuttering and to react on the '*Error'* signal immediately.

The behavior of the required system can be described precisely using an automaton that could measure time (Fig. 1). The automaton starts in state *N* and reads the input. If the train approaches, the input reads '*a*' and the automaton moves to state *T*. Now the input can stutter, but the automaton does not react to signal '*b*', until it has continued to be in state *T* at least through the period Δt. Afterwards, if '*b*' still holds, the automaton returns back to state *N* and continues as before. If the input reads '*Error*', the automaton moves to state *X*.

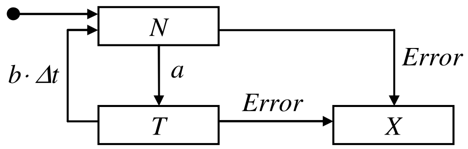


Fig. 1. Filtering device with detection of errors

The notation in Fig. 1 suggests that period Δt is attributed to a transition between states, rather than to a state. This is because a transition is enabled by a combination of an input symbol and a timer symbol.

Formal definition of the filtering device can be written as follows:

$S = \{ N, T, X \}$

$\Sigma = \{ a, b, Error \}$

$\Omega = \{ no\ approach, approach, don't\ know \}$

$\Gamma = \{ t_1 \}$

τ: $\tau(t_1) = (T, \Delta t)$

δ:	$\delta(N, a)=T$	$\delta(N, b)=N$	$\delta(N, Error)=X$
	$\delta(T, a)=T$	$\delta(T, b)=T$	$\delta(T, Error)=X$
	$\delta(T, a, t_1)=T$	$\delta(T, b, t_1)=N$	$\delta(T, Error, t_1)=X$
	$\delta(X, a)=X$	$\delta(X, b)=X$	$\delta(X, Error)=X$

$s_0 = N$

ω:	$\omega(N) = no\ approach$	$\omega(T) = approach$	$\omega(X) = don't\ know$

The granularity of time ε has not been defined in [4].

3.1 Execution of a Finite State Time Machine

Moore automaton models a device that cooperates with its environment. The execution of an automaton starts in state s_0. The environment generates a sequence of input symbols $a_0, a_1, \ldots, a_k, \ldots$ and the automaton moves through a sequence of states $s_0, s_1, \ldots, s_k, \ldots$ such that $s_{k+1} = \delta(s_k, a_k)$ for $k=0,1,\ldots$. Each state s_k of the automaton corresponds to an output symbol $q_k = \omega(s_k)$. This way the automaton responds to a sequence of input symbols $a_0, a_1, \ldots, a_k, \ldots$ with a sequence of output symbols $q_0, q_1, \ldots, q_k, \ldots$.

A finite state time machine adds to the model the dimension of time. The function τ defines for each state $s \in S$ a set of timers $T(s)$, such that:

$$T(s) = \{\, t \in \Gamma : (\exists r \in R^+)[\tau(t) = (s,r)] \,\}$$

A timer t, such that $\tau(t) = (s,r)$, will be denoted $t_{s,r}$. The same symbol will be used for the value of r, rounded up to the nearest multiplicity of ε. This will not lead to a misunderstanding, as the actual meaning of $t_{s,r}$ will always be clear from the context.

A nonempty set $T(s)$ can be ordered with respect to the value of $t_{s,r}$:

$$T(s) = \{\, t_{s,r1}, \ldots t_{s,rp} \,\}$$

The sequence $\{\, t_{s,r1}, \ldots\ t_{s,rp} \,\}$ defines a partition of time into $p+1$ intervals:

$$[0, t_{s,r1}) \ldots , [t_{s,rp}, \infty)$$

There is only one interval $[0,\infty)$ if the set $T(s)$ was empty. The execution of a finite state time machine starts in state s_0. When it enters a state $s \in S$, it enters the first time interval $[0, t_{s,r1})$ as well. The machine executes in this interval by taking an input symbol a and moving from the current state s to the next state $s' \in S$, such that $s' = \delta(s,a)$.

If the machine has continued to be in state s at least through $t_{s,r1}$ time units, then it moves to the time interval $[t_{s,r1}, t_{s,r2})$, then to $[t_{s,r2}, t_{s,r3})$ and so on. The machine executes in a time interval $[t_{s,rj}, t_{s,r(j+1)})$ by taking an input symbol a and moving from the current state s to the next state $s' \in S$, such that $s' = \delta(s,a,t_{s,rj})$. The flow of time does not depend on the sequence of input symbols. Therefore, unlike in a Moore automaton, there is no deterministic mapping from an input sequence $a_0, a_1, \ldots, a_k, \ldots$ to an output sequence $q_0, q_1, \ldots, q_k, \ldots$. The response of a finite state time machine depends on the time intervals within the sequence of input symbols.

3.2 Relation to Other Models

Moore automaton is equivalent to a finite state time machine with no timers, i.e.:

- $(\forall s \in S)[card(T(s)) = 0]$

If the above condition holds, then no partitioning of time exists and there is a single time interval $[0,\infty)$ for each state of the automaton.

PLC-automaton [4] can be converted into a finite state time machine such that:

- $(\forall s \in S)[card(T(s)) \leq 1]$

Let $P=(S, \Sigma, \delta, s_0, \varepsilon, St, Se, \Omega, \omega)$ be a PLC-automaton. The equivalent finite state time machine $A=(\underline{S}, \underline{\Sigma}, \underline{\Gamma}, \underline{\tau}, \underline{\delta}, \underline{s_0}, \underline{\varepsilon}, \underline{\Omega}, \underline{\omega})$ can be constructed as follows:

- The elements $\underline{S}, \underline{\Sigma}, \underline{s_0}, \underline{\varepsilon}, \underline{\Omega}, \underline{\omega}$ of A are equal to $S, \Sigma, s_0, \varepsilon, \Omega, \omega$ of P, respectively.
- The other three elements of A are the following:

$\underline{\Gamma} = \bigcup_{s\in S:\ St(s)>0} \{ t_{s,St(s)} \}$ – note that this defines the timer function $\underline{\tau}$, as well.

$$\underline{\delta}(s,a) = \begin{cases} s & \text{if } St(s)>0 \text{ and } a\in Se(s) \\ \delta(s,a) & \text{otherwise} \end{cases}$$

$$\underline{\delta}(s,a,t_{s,St(s)}) = \delta(s,a)$$

Timed automaton [3] represents a broader class of models than finite state time machine. This is because a clock of a timed automaton can measure time between two arbitrary transitions in a state transition graph, while the life of a timer of a finite state time machine is limited to the period of time, in which the machine continues being in a single state. A minor drawback of timed automaton is the lack of output symbols.

The advantages of the model of a finite state time machine are simplicity, expressiveness and ease of implementation. The limitation on the life of timers helps in keeping the size of the state space small. To demonstrate the expressive power of the model, consider a requirement to measure the time period between two events: '*a*' and '*b*', and to classify the delay as: *Short*, *middle*, *long* or *infinite*. The problem cannot be represented directly as a PLC-automaton [4]. Finite state time machine model (Fig. 2) is simple and understandable.

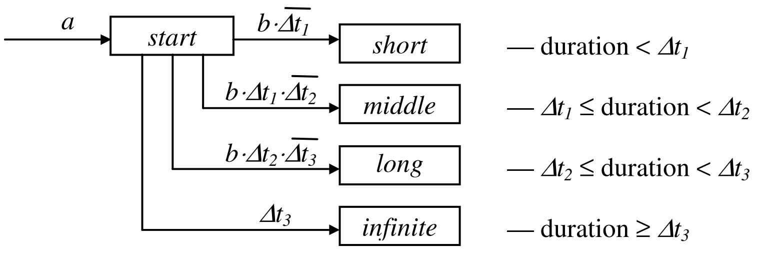

a
start
$b\cdot\overline{\Delta t_1}$
short
— duration < Δt_1
$b\cdot\Delta t_1\cdot\overline{\Delta t_2}$
middle
— $\Delta t_1 \le$ duration < Δt_2
$b\cdot\Delta t_2\cdot\overline{\Delta t_3}$
long
— $\Delta t_2 \le$ duration < Δt_3
Δt_3
infinite
— duration $\ge \Delta t_3$

Fig. 2. Measurement and classification of time periods ($0 < \Delta t_1 < \Delta t_2 < \Delta t_3$)

The semantics of a finite state machine can be defined formally using the Duration Calculus [5], which is a real-time extension to the discrete interval temporal logic. Such a definition allows formal reasoning and proving correctness of the model. This topic is not covered by this paper.

4 Code Generation

A developer defines the algorithm of processing input signals into output signals of a PLC by means of a state diagram. The semantics of this model is defined formally by a finite state time machine. Input and output symbols of the machine correspond to particular combinations of input or output signals of the PLC, respectively. Timer symbols correspond to timers, i.e. program elements that can measure time and signal the expiration of time periods that are defined by the timer function.

The states of the finite state time machine are stored within the controller as states of flip-flops, used by a program. Using *n* flip-flops one can store at most 2^n states. The mapping of states into the states of flip-flops (coding of states) is not unique, and can be a result of a design decision or an optimization procedure. A program of the controller implements the transition function in such a way that each pass through the program fires a single transition between the states of the finite state time machine.

PLC program takes the form of a ladder diagram [1] and is structured into a sequence of lines, each of which describes a Boolean condition to set or reset a flip-flop, a timer or an output signal, according to the values of input signals, states of flip-flops and timers. The Boolean conditions reflect the selected coding of states and implement the transition function, the output function and the timer function.

The development process of a PLC program consists of two phases. The first phase, which corresponds to the requirements modeling and analysis, followed by a safety analysis, requires creativity of the developer and must be performed manually. The second phase, which corresponds to program design and implementation, can be performed automatically. The first phase is not covered by this paper, but a detailed discussion can be found in [6,7].

Design and implementation phase starts with the verified model of a state diagram developed previously. The phase consists of the six basic steps:

1. Coding states.
2. Implementing the timer function.
3. Implementing the transition function.
4. Filling up the state space (error recovery).
5. Implementing the output function.
6. Building a program.

A description of the conversion process from a state diagram to a ladder diagram, given in the reminder of Sect. 4, will be illustrated using a case study of a simple bottling line. Although simply stated, the problem contains many of the elements of actual control systems.

4.1 Case Study

A bottling line (Fig. 3) consists of a bottle supply with a gate, a conveyor system, a scale platform and a bottle-filling pipe with a valve. Bottles to be filled are drawn one by one from the supply of bottles and moved to the scale platform by the conveyor. As soon as the bottle is at required position, a contact sensor attached to the platform is depressed and the bottle-filling valve is opened. The scale platform measures the weight of the bottle with its contents. When the bottle is full, the bottle-filling valve is shut off, and an operator manually removes the bottle from the line. Removing the bottle releases the contact sensor, and the entire cycle repeats automatically.

The current line status is described by a set of two-state signals issued by the plant sensors and switches:

S start the line: A manual signal that enables the repetitive line operation;
P suspend the line: A manual signal that suspends temporarily the bottling process;
R bottle ready: A signal from the electrical contact of the platform sensor;
F bottle full: A signal issued by the scale.

The controller reads the current line status and yields the three control signals:

G open the gate of the bottle supply (a pulse signal of the length Δt_1),
T start the conveyor,
Z open the bottle-filling valve.

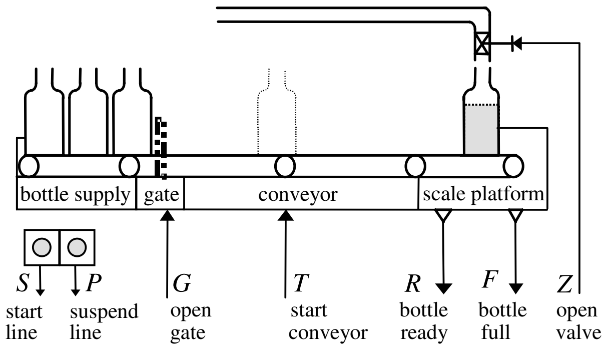


Fig. 3. Bottling line

There are three different modes of control of the bottling line: *Working* (regular line operation), *Blocked* (when something went wrong) and *Suspended* (a maintenance mode). Different modes of control are modeled as different states in a state diagram (Fig. 4). *Working* mode is modeled as a super-state, which has four sub-states nested that correspond to the particular phases of the bottling process.

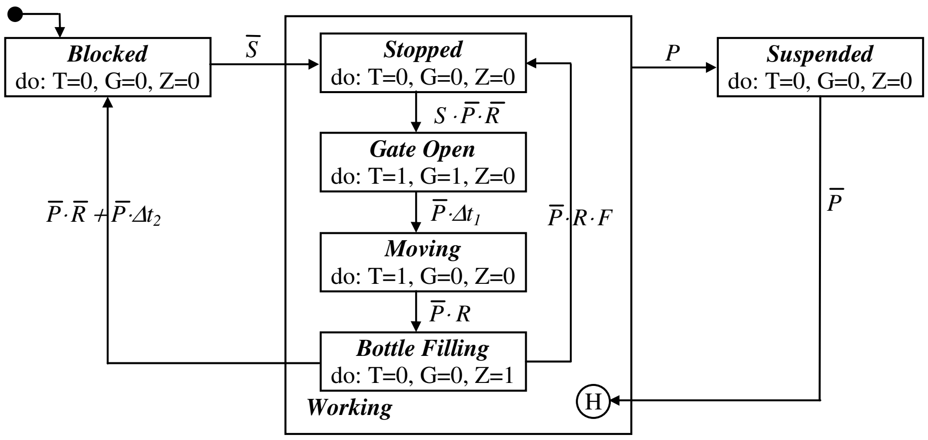


Fig. 4. Optimized state diagram of a bottling-line

The model defines the desired behavior of the bottling line. It implements a safety feature that the line is blocked until a manual intervention (confirmed by depressing of *S*), if the bottle on the scale platform was broken or the bottle-filling phase has not been finished within the period of Δt_2. The process of building the requirements specification, safety analysis and the optimization of the model, has been described and discussed in detail in [7].

4.2 Coding States

The algorithm for coding states in a hierarchical state diagram traverses the hierarchy in a top-down manner and assigns a separate group of flip-flops to code the sub-states of each super-state. This way, at least two flip-flops are needed to code the three states at the top level in Fig. 4, and the next two flip-flops are used to code the sub-states within the state *Working*. A selected coding of states is shown in Table 1.

Table 1. The coding of states (flip-flops: *M1*, *M2*, *M3*, *M4*)

M1	*M2*	*M3*	*M4*	Bottling line state
0	0			*Blocked*
1	0	0	0	*Stopped*
1	0	0	1	*Gate Open*
1	0	1	1	*Moving*
1	0	1	0	*Bottle Filling*
1	1	*	*	*Suspended*

There are six states at the lowest level of nesting shown explicitly in Fig. 4 and listed in Table 1. However, the history indicator adds an additional implicit memory of the former sub-states of the state *Working* that are to be re-entered from *Suspended.* Hence, there are in fact four sub-states nested in the state *Suspended* that correspond to sub-states of the state *Working*. These sub-states preserve the same coding of *M3* and *M4*. The two transitions between *Working* and *Suspended* in Fig. 4 are then transformed into a set of four pairs of transitions between the corresponding sub-states. The semantics of history indicator and the transformation described above is defined formally in Sect. 4.3. One can note that the coding of states presented in Table 1 is more economical than the one-hot coding used routinely for hardwired controllers.

4.3 Formal Model

There are nine states, sixteen input symbols and two timers in the finite state machine, which defines the semantics of the state diagram in Fig. 4. These sets together with the timer function and the transition function are defined below:

$$S = \{\ Blocked,\ Stopped,\ GateOpen,\ Moving,\ BottleFilling,$$
$$Suspended\text{-}Stopped,\ Suspended\text{-}Open,\ Suspended\text{-}Moving,\ Suspended\text{-}Filling\ \}$$
$$\Sigma = \{\ S\cdot P\cdot R\cdot F\ ,\ S\cdot P\cdot R\cdot \overline{F}\ ,\ S\cdot P\cdot \overline{R}\cdot F\ ,\ S\cdot P\cdot \overline{R}\cdot \overline{F}\ ,\ \ldots,\ \overline{S}\cdot \overline{P}\cdot \overline{R}\cdot \overline{F}\ \}$$
$$\Gamma = \{\ t_1,\ t_2\ \}$$
$$\tau:\ \tau(\ t_1\ ) = (GateOpen,\ \Delta t_1\ ) \qquad \tau(\ t_2\ ) = (BottleFilling,\ \Delta t_2\ )$$

δ: $\delta(Blocked, \overline{S}) = Stopped$

$\delta(Stopped, P) = Suspended\text{-}Stopped$ $\quad$ $\delta(Stopped, S\cdot\overline{P}\cdot\overline{R}) = GateOpen$

$\delta(GateOpen, P) = Suspended\text{-}Open$ $\quad$ $\delta(GateOpen, P, t_1) = Suspended\text{-}Open$

$\delta(GateOpen, \overline{P}, t_1) = Moving$

$\delta(Moving, P) = Suspended\text{-}Moving$ $\quad$ $\delta(Moving, \overline{P}\cdot R) = BottleFilling$

$\delta(BottleFilling, P) = Suspended\text{-}Filling$ $\quad$ $\delta(BottleFilling, P, t_2) = Suspended\text{-}Filling$

$\delta(BottleFilling, \overline{P}\cdot R\cdot F) = Stopped$

$\delta(BottleFilling, \overline{P}\cdot\overline{R}) = Blocked$ $\quad$ $\delta(BottleFilling, \overline{P}, t_2) = Blocked$

$\delta(Suspended\text{-}Stopped, \overline{P}) = Stopped$ $\quad$ $\delta(Suspended\text{-}Open, \overline{P}) = GateOpen$

$\delta(Suspended\text{-}Moving, \overline{P}) = Moving$ $\quad$ $\delta(Suspended\text{-}Filling, \overline{P}) = BottleFilling$

In all other cases $\delta(s, a) = s$ and $\delta(s, a, t) = s$. These transitions are not shown in Fig 4. The usual Boolean notation for the subsets of input symbols is used in the above definition of the function δ, e.g.: $S\cdot\overline{P}\cdot\overline{R}$ represents the set $\{S\cdot\overline{P}\cdot\overline{R}\cdot F, S\cdot\overline{P}\cdot\overline{R}\cdot\overline{F}\}$.

4.4 Implementing Timers

Each timer symbol of a finite state time machine is implemented within a PLC controller by a separate timer block of a ladder diagram. A timer block is a conceptual device that has one input signal, which can set (enable) the timer, and one output signal. As long as the input signal equals **0**, the timer is reset with the output equal to **0**. When the input signal changes to **1**, the timer is set and starts counting time. The output signal changes to **1** as soon as the input signal has continued to be **1** for a predefined period of time. Such type of a timer block is called a *delay on make flip-flop*.

A Boolean condition that sets a timer depends on the coding of this state, which is assigned to the timer by the timer function. For example, timers t_1 and t_2 (See Sect. 4.2), are assigned to states *Gate Open* and *Bottle Filling*, respectively. Hence, the conditions to set the timers can be read from Table 1:

$$\text{Set } t1 = M1\cdot\overline{M2}\cdot\overline{M3}\cdot M4$$
$$\text{Set } t2 = M1\cdot\overline{M2}\cdot M3\cdot\overline{M4}$$

Each time the above two expressions are executed by a PLC, time is counted and the outputs of the timers are set appropriately.

4.5 Implementing the Transition Function

The transition function of a finite state time machine defines conditions to set or reset flip-flops. It is implemented by a sequence of Boolean expressions that depend on the coding of states, input signals and timers.

Consider the transition from *Blocked* to *Stopped* in Fig. 4, described formally as: $\delta(Blocked, \overline{S}) = Stopped$. The transition occurs in a state such that $M1=\mathbf{0}$ and $M2=\mathbf{0}$ (Table 1), when $S=\mathbf{0}$. After the transition has occurred, the state changes to the one, in which $M1=\mathbf{1}$ and $M2=\mathbf{0}$ and $M3=\mathbf{0}$ and $M4=\mathbf{0}$. So, the transition is implemented by setting *M1* flip-flop and resetting *M3*, *M4*:

$$\text{Set } M1 = \overline{S}\cdot\overline{M1}\cdot\overline{M2}$$
$$\text{Res } M3 = \overline{S}\cdot\overline{M1}\cdot\overline{M2}$$
$$\text{Res } M4 = \overline{S}\cdot\overline{M1}\cdot\overline{M2}$$

Similarly, to implement the transition from *Bottle Filling* to *Blocked* one must reset *M1* (*M2* is reset in *Bottle Filling*, the values of *M3* and *M4* are insignificant). Hence:

$$\text{Res}\, M1 = (\overline{P} \cdot \overline{R} + \overline{P} \cdot t2) \cdot M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$$

Boolean expressions that implement the other transitions by setting and resetting particular flip-flops can be defined similarly.

In order to ensure the atomicity of transitions, a set of secondary flip-flops can be used, assigned on a one-to-one basis to the primary flip-flops that are used to encode the system states. The secondary flip-flops store the next state of the system, calculated by the execution of Boolean expressions, until the computation of all the expressions has been finished. The next state is then converted into the current state by copying the state of secondary flip-flops to the primary flip-flops [6].

There are four flip-flops in Table 1. Denote the secondary flip-flops: *M11* ... *M14*, respectively. A complete sequence of Boolean expressions that implement the transition function can be defined as follows:

(a1) $\text{Set}\, t1 = M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4$
(a2) $\text{Set}\, t2 = M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$
(b1) $\text{Set}\, M11 = \overline{S} \cdot \overline{M1} \cdot \overline{M2}$
(b2) $\text{Set}\, M12 = P \cdot M1 \cdot \overline{M2}$
(b3) $\text{Set}\, M13 = \overline{P} \cdot t1 \cdot M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4$
(b4) $\text{Set}\, M14 = S \cdot \overline{P} \cdot \overline{R} \cdot M1 \cdot \overline{M2} \cdot \overline{M3} \cdot \overline{M4}$
(b5) $\text{Res}\, M11 = (\overline{P} \cdot \overline{R} + \overline{P} \cdot t2) \cdot M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$
(b6) $\text{Res}\, M12 = \overline{P} \cdot M1 \cdot M2$
(b7) $\text{Res}\, M13 = \overline{P} \cdot R \cdot F \cdot M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4} + \overline{S} \cdot \overline{M1} \cdot \overline{M2}$
(b8) $\text{Res}\, M14 = \overline{P} \cdot R \cdot M1 \cdot \overline{M2} \cdot M3 \cdot M4 + \overline{S} \cdot \overline{M1} \cdot \overline{M2}$
……………………………………………
(d1) $M1 = M11$
(d2) $M2 = M12$
(d3) $M3 = M13$
(d4) $M4 = M14$

It can be noted that the expressions to set timers have been placed in the sequence before the expressions that implement the transition function. A PLC controller samples all the input signals at the beginning of each program cycle, before executing any Boolean expressions. This way the input signals are up-to-date, but stable during the entire program execution. The output of a timer can also be used as an input to expressions. Therefore the timers are processed before the computation of expressions that implement the transition function can start.

The functions can be minimized using the standard rules of Boolean algebra, e.g.:

(b5) $\text{Res}\, M11 = (\overline{R} + t2) \cdot \overline{P} \cdot M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$

4.6 Filling Up the State Space

The requirement for high dependability of a control program cannot be fulfilled without a planned reaction to faults that can develop during the program execution.

Consider a failure that sets an improper value of a primary flip-flop and leads to a faulty combination that does not correspond to any valid state of the system [6]. Such a combination of flip-flops that is not a valid system state can be detected automatically. Then, a policy of state recovery must exist, which returns the system to a valid and safe state. One choice of such a policy is entering a *safe stop* position, if the one exists in the application domain. To implement such a policy, the next state of the system must be verified before becoming the current state. If the computed next state appears invalid, it is discarded, and the system is stopped in a safe position.

In order to implement such a fail-stop strategy an auxiliary flip-flop can be set temporarily if a faulty next state was observed. If this is the case, the primary flip-flops are reset to the *safe stop* position, in which the automaton waits for being restarted.

In the bottling line example, all the combinations of flip-flops such that $M1 = \mathbf{0}$ and $M2 = \mathbf{1}$ are not used and do not correspond to any valid state. On the other hand, a safe stop position exists and corresponds to the state *Blocked*. A sequence of Boolean expressions that implement the fail-stop strategy can be written as follows:

(c1) $M5 = \overline{M11} \cdot M12$

(d1) $M1 = M11 \cdot \overline{M5}$

(d2) $M2 = M12 \cdot \overline{M5}$

(d3) $M3 = M13 \cdot \overline{M5}$

(d4) $M4 = M14 \cdot \overline{M5}$

4.7 Implementing the Output Function

The output function defines conditions to set or reset the output signals in relation to the current state of the finite state time machine. It is implemented by a sequence of Boolean expressions that depend on the coding of states. These expressions must be computed at the end of the program cycle, after copying the state of secondary flip-flops to the primary flip-flops – this way the physical outputs of a PLC will be set consistently with the computed current state of the system as soon as possible.

A complete sequence of Boolean expressions that implement the output function of the bottling line controller can be defined as follows:

(e1) $G = M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4$

(e2) $Z = M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$

(e3) $T = M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4 + M1 \cdot \overline{M2} \cdot M3 \cdot M4 = M1 \cdot \overline{M2} \cdot M4$

4.8 Building a Program

The sequence of Boolean expressions, generated by an automatic tool from a state diagram, or a set of state diagrams, defines in all detail a program for a PLC. Such a program can be expressed in the language of a ladder diagram or an instruction list [1,8]. Each expression is converted into a single line of the ladder. Disjunction of terms is represented by parallel branches within the line, while conjunction of symbols is represented by serial elements within a given branch. Negation of an argument is implemented by a normally closed contact. Each timer symbol is implemented by a separate timer provided by the language. A part of the program for a bottling-plant controller is shown in Figure 5.

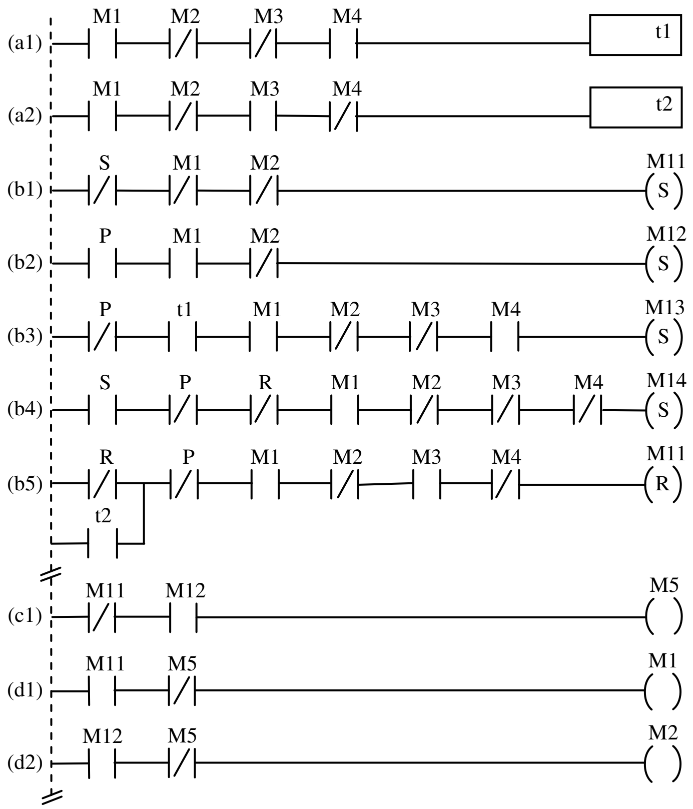


Fig. 5. A part of the program of a bottling line controller

Finite state time machine can also be implemented using a procedural language, e.g. C. A description of the conversion process is outside the scope of this paper.

5 Conclusions

This paper describes a method for automatic generation of code for PLC controllers. The process of converting a specification into a program code is defined formally, using a model of finite state time machine. The method has the following advantages:

- Graphical requirements specification, based on a subset of UML state diagrams.
- Formal model of the specification with formally defined meaning and behavior.
- Formal definition of the conversion process.
- The potential for formal analysis using a temporal logic of Duration Calculus.

A disadvantage is complexity that results from exponential growth of the sets of input symbols and states. However, the concept of input symbol helps in making the specification unambiguous, and the concepts of hierarchical state diagram and history indicator make part of the state space invisible to the modeler. The full size of the state space appears only at the level of a finite state time machine. Appropriate representation can make automatic verification of systems of 10^{20} states feasible [9].

Models and methods that are based on the theory of finite state machines are recommended by IEC for modeling and developing safety related systems [10].

The method has been tested manually in a student lab, where the students had to develop programs (ladder diagrams) executed finally by Siemens S7 PLC controllers. The method worked well, and the model of a state diagram proved to be well suited to the education profile of the software engineering students working in the lab.

A version of an automatic tool for program generation is currently being tested. The tool inputs UML state diagrams, generated by Rational Rose [11], and outputs the ladder diagrams for Siemens Step 7 programming environment [8]. A huge advantage of the tool over the prototype generation feature of Rational Rose itself, is simplicity – once a state diagram has been developed and validated, the generation process requires virtually no effort of the developer.

The plans for future work are aimed at the application of Duration Calculus formulae for proving real-time properties of finite state time machines. The other goal is to cover concurrent operations that are allowed in the UML-based state diagrams.

References

1. IEC 1131-3, Programmable controllers – part 3: Programming languages, IEC (1993)
2. Douglass, B.P.: Real-Time UML, Addison-Wesley, Reading, Massachusetts (1998)
3. Alur, R., Dill, D.L.: Automata-theoretic verification of real-time systems. In: Formal Methods for Real-Time Computing, Trends in Software Series. John Wiley & Sons (1996) 55-82
4. Dierks, H.: PLC-Automata: A New Class of Implementable Real-Time Automata. In: Bertran, M., Rus, T. (eds.): Transformation-Based Reactive Systems Development. LNCS, Vol. 1231. Springer-Verlag, Berlin (1997) 111-125
5. Chaochen, Z.: Duration Calculi: An overview, LNCS, Vol. 735, Springer-Verlag, Berlin (1993) 256-266
6. Sacha, K.: A Simple Method for PLC Programming. In: Colnaric, M., Adamski, M., Węgrzyn, M. (eds): Real-Time Programming 2003, Elsevier (2003) 27-31
7. Sacha, K.: Dependable Programming Using Statechart Models, Proc. 29th IFAC Workshop on Real Time Programming, Istanbul (2004)
8. Siemens, SIMATIC S7-200 Programmable Controller, System manual, Siemens (1998)
9. Burch, J.R., Clarke E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation, Vol. 98,2 (1992) 142-170
10. IEC 61508, Functional Safety: Safety-Related Systems, IEC, 1998/2000
11. Rational Rose Corporation, http://www.rational.com/product/rose

The TACO Approach for Traceability and Communication of Requirements

Terje Sivertsen[1], Rune Fredriksen[1], Atoosa P-J Thunem[1],
Jan-Erik Holmberg[2], Janne Valkonen[2], Olli Ventä[2], and Jan-Ove Andersson[3]

[1] Institute for Energy Technology, Software Engineering Laboratory, PB 173,
NO-1751 Halden, Norway
{terje.sivertsen, rune.fredriksen, atoosa.p-j.thunem}@hrp.no
[2] VTT Industrial Systems, P.O. Box 1301, FIN-02044 VTT, Finland
{jan-erik.holmberg, janne.valkonen, olli.venta}@vtt.fi
[3] Ringhals AB, Barsebäck Kraft, P.O. Box 524, SE-246 25, Löddeköpinge, Sweden
jan-ove.andersson@ringhals.se

Abstract. This paper outlines the main achievements of the TACO project. The overall objective of the TACO project was to improve the knowledge about principles and best practices related to the issues concretised in the TACO preproject. On the basis of experiences in the Nordic countries, the project aimed at identifying the best practices and most important criteria for ensuring effective communication in relation to requirements elicitation and analysis, understandability of requirements to all parties, and traceability of requirements through the different design phases. It is expected that the project will provide important input to the development of guidelines and establishment of recommended practices related to these activities.

1 Introduction

The title of the reported project is "Traceability and Communication of Requirements in Digital I&C Systems Development", abbreviated TACO. The project was funded by Nordic nuclear safety research (NKS) and the project number was NKS_R_2002_16.

The overall objective of the TACO project was to improve the knowledge on principles and best practices related to the issues concretised in the preproject. On the basis of experiences in the Nordic countries, the project aimed at identifying the best practices and most important criteria for ensuring effective communication in relation to requirements elicitation and analysis, understandability of requirements to all parties, and traceability of requirements through the different design phases. It is expected that the project will provide important input to the development of guidelines and establishment of recommended practices related to these activities.

The overall aim of the first phase of the project, the TACO preproject, which was carried out in the second half of 2002, was to identify the main issues related to traceability and communication of requirements in digital I&C systems development. By focusing on the identification of main issues, the preproject provided a basis for prioritising further work, while at the same time providing some initial recommendations related to these issues. The establishment of a Nordic expert network within the subject was another important result of the preproject.

The project activities in 2003 constituted a natural continuation of the preproject, and focused on the technical issues concretised in the preproject report. The work concentrated on four central and related issues, viz.

- Representation of requirements origins
- Traceability techniques
- Configuration management and the traceability of requirements
- Identification and categorisation of system aspects and their models

The results from the preproject and the activities in 2003 were presented at the first TACO Industrial Seminar, which took place in Stockholm on the 12th of December 2003. The seminar was hosted by the Swedish Nuclear Power Inspectorate (SKI).

In 2004, the work has focused on providing a unified exposition on the issues studied and thereby facilitating a common approach to requirements handling, from their origins and through the different development phases. Emphasis has been put on the development of the TACO Traceability Model. The model supports understandability, communication and traceability by providing a common basis, in the form of a requirements change history, for different kinds of analysis and presentation of different requirements perspectives. Traceability is facilitated through the representation of requirements changes in terms of a change history tree built up by composition of instances of a number of change types, and by providing analysis on the basis of this representation. Much of the strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation.

The work was presented at the second TACO Industrial Seminar, which took place in Helsinki on the 8th of December 2004. The seminar was hosted by the Finnish Radiation and Nuclear Safety Authority (STUK).

2 The TACO Approach

The present chapter introduces the TACO common approach to requirements handling, called the TACO Shell. The idea is that the shell is a framework for traceability and communication of requirements, which can be filled with different contents to reflect the needs in different application areas. To facilitate its practical use, the TACO Shell is provided with guidelines, comprising ingredients and recipes, for filling and utilizing the TACO Shell. The TACO approach to requirements change management is based on a mathematically well-founded traceability model, called the TACO Traceability Model, where the introduction, changes, and relationships between different requirements, design steps, implementations, documentation, etc. are represented in terms of an extended change history tree. The traceability model adopted aims at forming the logic needed for formalising the activities related to change management and hence their further automation. By complementing the model with appropriate terminology, data structures and guidelines for use, the model can be adapted to the different needs related to management of changes in computer-based systems, including safety-critical and security-critical systems.

2.1 The TACO Shell

The TACO Shell is the overall TACO framework for requirements handling, and represents a generic approach to lifecycle-oriented, traceability-based requirements management. The TACO Shell comprises the overall methodology, the TACO Traceability Model, and the different guidelines related to its contents (ingredients) and use (recipes). By varying the ingredients and recipes, the shell can be used for the development of different kinds of target systems, with different requirements origins, different emphasis on quality attributes, and different selection of dependability factors.

2.2 The TACO Traceability Model

The TACO Traceability Model adopts several of the ideas to fine-grained traceability presented in [3]. Accordingly, traceability is facilitated by representing the requirements changes in terms of a change history tree built up by composition of instances of seven different change types, and to provide analysis on the basis of this representation. The change types correspond to the following generic actions performed on requirements, or more generally, paragraphs (from [3]):

- Creating a new paragraph with no prior history.
- Deleting an existing paragraph.
- Splitting an existing paragraph, thereby creating a number of new paragraphs.
- Combining existing paragraphs by a new paragraph.
- Replacing existing paragraphs by a new paragraph.
- Deriving a new paragraph from existing paragraphs.
- Modifying a paragraph without changing its meaning.

The change history can be represented by a tree where the paragraphs constitute the nodes. The tree representation constitutes an appropriate basis for different kinds of analysis, including finding

- all initial paragraphs;
- all deleted paragraphs;
- all applicable paragraphs;
- the complete history of a paragraph;
- the complete backwards traceability from a set of paragraphs;
- the complete forwards traceability from a set of paragraphs;
- the legality of a proposed requirements change.

The possibility to find the backwards or forwards traceability from a set of requirements facilitates backwards and forwards branch isolation and analysis of the change history. The versatility of the representation can be further improved by extending the representation of the paragraphs to include different parameters that classify the requirements, provide additional information, etc. Possible parameters are discussed later in the report.

When it comes to the representation of the actual parameters, it is important to distinguish between (1) the information that is essential to identify the paragraph, and (2) the various information associated to this parameter. Conceptually, and from a perspective of modularity, it is useful to let the nodes in the change history tree represent the necessary and sufficient information related to the identity of a paragraph. In the TACO Traceability Model, a paragraph is represented by the combination of a unique identifier for this paragraph and a version number to distinguish several versions of the same paragraph. At any time, only the latest version of a paragraph can be an applicable paragraph. That is, a new version of a paragraph is introduced only if this replaces old versions. In any case, it is possible to make duplicates of a paragraph when these are treated as different paragraphs. This can also be used for representing different variants of the same requirement, possibly with "application conditions" attached as guidelines to every single variant. Each variant will however be represented with a separate paragraph.

It is important to note that concepts similar to those described above for the TACO Traceability Model can be found in commercial tools for version control and configuration management. Although the change types might have other names, they typically resemble those defined here. In general, however, these tools do not offer an identifiable, formally defined traceability model, and leave to the user to define the actual semantics underlying the different change types. The strength of the TACO Traceability Model is that it aims at forming the logic needed for formalising the activities related to change management and hence their further automation.

Conceptually, we can think of a node of the change history tree as a versioned paragraph, represented by a pair of a paragraph identifier and a version number. In the following we will use the change history in Figure 1 as an example.

The development of the requirements in Figure 1 starts with the introduction of the paragraphs p1, p2, and p3. At later stages, another two new paragraphs are introduced, viz. p5 and p11. All the other paragraphs are developed on basis of these five paragraphs. Paragraphs p1 and p2 are first modified and then combined into a new paragraph p4. After a modification, this paragraph is split into four separate paragraphs p7 to p10. The latter of these paragraphs is modified and then combined with p6, originally derived from paragraphs p3 and p5, giving paragraph p12. Note that, at any point in the development of the paragraphs, at most one version of a paragraph is applicable (in the sense that it is the valid version of the paragraph). It is certainly possible to represent the change history tree textually in such a way that the temporal relationships between the different changes are maintained.

Let us now consider the other information attached to a paragraph. As has been argued in the foregoing, it is not necessary to represent this information in the change history tree. The purpose of the tree is to give a complete representation of the changes and how they are related to each other. What about the other information, including the actual text of the paragraph? Formally, we can think of these relations in terms of some basic mathematical concepts:

- Sets: These are finite collections of objects of some type, and can be used for representing subsets of the paragraphs. By way of example, the classification of paragraphs with respect to Business plan, Requirements document, Design specification, etc, can be represented by means of separate, maybe overlapping sets corresponding to the different classification terms. Finding, say, all Business plan related requirements is then trivial, since they are given by the corresponding set. Checking whether a requirement belongs to the Business plan is also easy and can

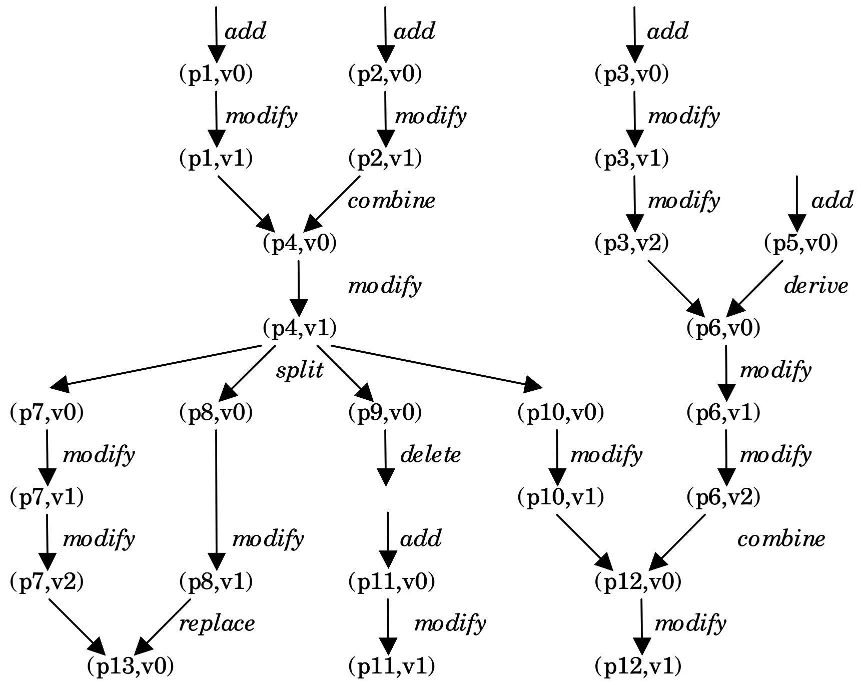


Fig. 1. The example change history

be done simply by checking whether the given paragraph is a member of the corresponding set. On the other hand, finding the class of a given paragraph cannot be done by simple look-up but involves checking all the different sets for membership.

- Mappings: These are functions from a source set to a target set, and can be used for assigning information to the paragraphs in a simple look-up fashion. With this solution, e.g. the classification of paragraphs can be represented by mappings from the paragraphs to their classification. Finding the classification of a requirement is then simple, since it reduces to looking up the classification of that requirement. Finding all requirements is possible, but less trivial than for sets, as it involves selecting all requirements that are mapped to a certain term. On the other hand, the concept of relation is more convenient if there may be more than one class for a requirement.
- Relations: These are more general than mappings, since they allow an element in the source set to be associated to more than one element in the target set. With this solution, finding the classification of a requirement involves finding all elements in the target set (the classes) that are related to the given requirement. Finding all requirements related to a certain class can alternatively be understood as the inverse relation.

Sets can be considered as being implemented as simple lists. Mappings and relations can be considered as being implemented as tables. As we will see in the continued discussion, these representation concepts will suffice for representing all information associated to the requirements. It is of course possible to represent the same information in other ways as well, as long as consistency is maintained.

A basic piece of information related to a requirement is certainly the statement (phrasing) of the requirement. Assuming that (at most) one statement is associated to each requirement, we may think of this information as being available by means of a mapping from versioned requirements to their statements, see Figure 2.

Table 1. Mapping from requirements to their statements

Requirement	Statement
(p1,v0)	<Statement of version v0 of paragraph p1>
(p1,v1)	<Statement of version v1 of paragraph p1>
(p2,v0)	<Statement of version v0 of paragraph p2>
...	...
(p13,v0)	<Statement of version v0 of paragraph p13>

As is evident from Table 1, the statement of a given requirement can be found by simple look-up in the table implementing the mapping. The table can be utilized in different ways. By way of example, finding all relevant requirements can be found by filtering the mapping with respect to the applicable paragraphs to find the subset of the mapping that relates to applicable paragraphs only. Filling in the relevant information is an obvious task of an information system designed to support the use of the model.

Other useful information can be represented in the same way. By way of example, a recurrent problem with modernization projects is the difficulties of recapturing both the “what” and the “why” of a requirement. In the TACO Traceability Model, the “what” is covered by Table 1. In a similar way, the “why” of the requirements can be covered by a similar mapping from requirements to comments giving information on the background, motivation, reasons, etc. for including the requirements.

2.3 Utilization

A possible utilization of the TACO Traceability Model is in the identification of relative influences, correlations, and conflicts between safety/security countermeasures and other dependability factors. On this basis, guidelines to the use, implementation, and verification of the different change types can be developed. These guidelines would have to reflect the identified relative influences, correlations, and conflicts in the sense that they provide a better basis for controlling the effects of changes. The guidelines should include descriptions on how different techniques can be applied for this purpose, such as the use of formal specification and proof for demonstrating the correct derivation of requirements, coding standards for implementation of specific design features, etc.

The utilization and applicability of the TACO Traceability Model will be further explored and documented by cooperation with other projects and partners. The results will be collected within the framework of the Nordic project MORE (Management of Requirements in NPP Modernization Projects).

On basis of compiled experiences on the problem of handling large amounts of information in relation to Nordic modernization projects, the MORE project will investigate how the approach to requirements management developed in the TACO project can be utilised to handle large amounts of evolving requirements in NPP modernization projects. It is expected that the activity will provide important input to the development of guidelines and establishment of recommended practices related to the management of requirements in such projects. This kind of input is of high importance to a common understanding between vendors, utilities, and regulators about the proper handling of requirements in the digital I&C systems development process, and consequently to the successful introduction of such systems in NPPs.

3 TACO Guidelines

The TACO project aims at providing input to the development of guidelines and establishment of recommended practices related to requirements elicitation and analysis, understandability of requirements to all parties, and traceability of requirements through the different design phases. In this chapter, guidelines will be presented to the practical use of the TACO Shell in activities related to the different lifecycle phases. The guidelines can be seen as comprising *ingredients* and *recipes* for filling and utilizing the TACO Shell. By gradually complementing the TACO Shell and the TACO Traceability Model with appropriate terminology, data structures and guidelines for use, the model can be adapted to the different needs related to management of changes in computer-based systems, including safety-critical and security-critical systems. By way of example, the model can organize communication and analysis of requirements by generating subsets of the change history showing the backwards or forwards traceability of given requirements. The TACO guidelines help to utilize these possibilities in practical work.

By varying the ingredients and recipes, the TACO Shell can be used for the development of different kinds of target systems, with different requirements origins, different emphasis on quality attributes, and different selection of dependability factors. The TACO guidelines can be developed on a continual basis to fit the use, implementation, and verification of the different change types. The guidelines should include descriptions on how different techniques can be applied, such as the use of formal specification and proof for demonstrating the correct derivation of requirements, coding standards for implementation of specific design features, etc.

The mathematical underpinnings of the TACO Guidelines is described in Sivertsen et al. [4] in terms of a functional specification of the change history tree, the different change types and different kinds of analysis that can be performed on basis of this representation. The TACO Traceability Model is specified in two layers, reflected in a hierarchy of two specifications. In the "lower" specification, the different change types are specified inductively as generators, thus providing a data structure for the change history. At the top of this specification, the different change types are specified as operators, checking that the given change is legal and producing a lower layer representation together with the set of applicable paragraphs. By applying these operators, only legal change histories are represented.

3.1 Validity of Requirements Changes

Software development needs to deal with changes to the requirements, also after the requirements specification phase ideally is completed and the requirements frozen. The evolutionary nature of software implies that changes will have to be anticipated. Other changes may be necessary due to our evolving understanding about the application under development.

One of the lessons learned in the software engineering area is that software should be designed for change. The focus in the present report is on how to manage the evolution of the requirements in this situation. The present section deals with how the TACO Traceability Model can be utilized in the validation of the changes representing this evolution.

The TACO Traceability Model is based on a number of change types that can be employed to manage requirements changes throughout the life cycle of a system. Each change introduced in the life cycle should in principle be validated. Depending on the level of rigidity or formality employed, the validity of a change can be done in a variety of ways, from a simple inspection to a formal mathematical proof. Notwithstanding these differences, we will in the following concentrate on what validity in general means for the different change types. Validity should not be confused with the legality of changes. While validation concerns the semantics of the changes, the legality of a change can be checked mechanically from the structure of the change history tree.

Creating: Applied on requirements, creating a new paragraph with no prior history involves introducing a new requirement. The validity of the requirement involves both its *correctness* with respect to its intended meaning, its *completeness* with respect to its coverage of its intended meaning, and its *consistency* with other requirements. In short, the validity of a new requirement requires that it faithfully reflects the intended meaning and that it is not in conflict with other requirements. This is the only change type that is allowed to introduce new requirements or new aspects of requirements that are not already covered by existing paragraphs.

Deleting: Deleting an existing paragraph involves that a requirement in fact is withdrawn from the set of requirements. A requirement can be deleted if either the requirement in itself is no longer valid, or it is covered by other requirements. To demonstrate the validity of the change therefore either involves showing that it is the intention to withdraw the requirement as such or showing that it can be derived from other requirements.

Splitting: Splitting an existing paragraph involves creating a number of new paragraphs that collectively replaces the given one. Applied on requirements, a paragraph split is valid only if the requirements given in the new paragraphs together cover the replaced requirement, but not more. In other words, splitting a paragraph is not valid if the new paragraphs require more or less than the replaced paragraph.

Combining: Combining a set of existing paragraphs involves creating a new paragraph on basis of the existing ones, without deleting any of the existing paragraphs. Applied on requirements, a combination of paragraphs is valid only if the new paragraph covers the given paragraphs, but not more. In other words, combining a set of paragraphs is not valid if the new paragraph requires more or less than the given paragraphs.

Replacing: Replacing a set of existing paragraphs involves creating a new paragraph that replaces the existing ones. The validity criterion is identical to that of combination. Replacing a set of paragraphs and splitting an existing paragraph are inverse changes.

Deriving: Deriving a new paragraph from a set of existing paragraphs involves creating a new paragraph on the basis of the existing ones, without deleting any of the existing paragraphs. Applied on requirements, deriving a new paragraph is valid only if the requirement is *one* of the possible results/consequences of the requirements it is derived from.

Modifying: Modifying a paragraph should involve no changes to its meaning. The new requirement should therefore cover the replaced requirement, but not more.

Attempts on demonstrating the validity of individual changes may reveal flaws in the requirements management, such as introducing new paragraphs in a paragraph split that actually adds new requirements that are not covered by the replaced requirement. Detecting such flaws can be utilized in the requirements change process to produce an appropriate requirements change history, such as specifying such added requirements in terms of separate changes of type *creating new paragraphs with no prior history*. Similarly, insufficient coverage of the replaced requirement in a split change can be made "clean" by complementing the split with separate changes of type *deleting an existing paragraph*. In this way, an invalid change can be replaced by a set of valid changes, and the need for demonstrating the validity of the different changes can be made explicit.

3.2 Formal Review and Test of Requirements

Due to the high costs associated with defects slipping through the requirements specification phase, formal review and test of the requirements documents are usually highly prioritised activities. Industrial experience shows that very often a significant fraction of the most critical software defects are introduced already in the requirements specification. Of this reason, it is generally recommended to carry out tests on this specification that are as near as exhaustive as possible, and for this purpose, the use of a formal approach is often advocated.

Requirements analysis and requirements validation have much in common, but the latter type of activity is more concerned with checking a final draft of the requirements document which includes all system requirements and where known incompleteness and inconsistency has been removed, see [1]. As such, it should be planned and scheduled in the quality plan for the project, and be carried out in accordance with good quality assurance practice.

One of the theses behind the present report is that the TACO Traceability Model can be used for revealing and correcting several kinds of shortcomings discovered during the validation of the requirements document. This is true in particular for problems related to lack of conformance with the standards employed. The validation of the requirements against a given standards can be carried out by utilizing the information included about the origins of the requirements.

Such a validation could include the following steps:

1. Add all the requirements from the given standard by creating new paragraphs. If certain requirements are found irrelevant, the exclusion of these can be made explicit by deleting these paragraphs. This also makes explicit the need to validate their exclusion.
2. Check that the applicable and deleted paragraphs together constitute the complete set of requirements given in the standard. This can partly be automated by keeping these requirements on file.
3. Validate the change history related to the applicable paragraphs originating from the standard, utilizing the guidelines listed in section 3.1.
4. Validate the deletion of paragraphs originating from the standard, utilizing the guidelines listed in section 3.1.

Using the TACO Traceability Model in validating the requirements document may be done in the context of a formal requirements review meetings, in accordance with general guidelines to such meetings. Requirements validation may also take other forms, like prototyping, model validation, and requirements testing, but the focus in the TACO project has been on the utilization of the requirements change history in the review meetings. For further reading on formal review meetings, see [1].

Requirements reviews are conventionally carried out as a formal meeting involving a group representing the stakeholders. The general idea is that the system stakeholders, requirements engineers and system designers together check the requirements to verify that they adequately describe the system to be implemented. Traceability and requirements changes are of course only part of the concern at such a meeting. The TACO Traceability Model may however provide important assistance for discovering requirements problems related to requirements conflicts or lack of conformance to standards and other requirements origins.

In the end, the requirements traceability is itself a concern of the requirements review. As discussed in [1], the requirements should be unambiguously identified, include links to related requirements and to the reasons why these requirements have been included. Furthermore, there should be a clear link between software requirements and more general systems engineering requirements. This relates to the obvious fact that the software engineering activity is part of the much larger systems development process in which the requirements of the software are balanced against the requirements of other parts of the system being developed [2]. Furthermore, the software requirements are usually developed from the more general system requirements, and thus the traceability and consistency with these requirements is a basic premise for a successful process and its resulting product.

3.3 Correctness of Implementation

The correctness of implementation is a quality that characterizes the ability of the application to perform its function as expected [2]. Reasoning about correctness therefore requires the availability of the functional requirements, and we say that the application is functionally correct if it behaves according to the specification of these requirements.

In principle, correctness is in this context a mathematical property that establishes the equivalence between the software and its specification. In practice, the assessment of correctness is done in a more or less systematic manner, depending on how rigorously the requirements are specified and the software developed. In any case, the assessment requires that the requirements can be traced forward to their implementation, and vice versa.

The TACO Traceability Model supports the assessment of correctness by relating the requirements and their implementation through the change history tree. This relationship can be utilized in both a forwards and backwards fashion. The TACO shell provides both forwards and backwards traceability analysis, without requiring separate links for forwards and backwards traceability. The different types of analysis can be defined on the basis of one and the same representation of the change history tree.

In general, a forward traceability approach to assessment of correctness would take the specified requirements as starting point, and then demonstrate that all the requirements have been correctly implemented. Analogously, a backward traceability approach would take the implementation as starting point and check the consistency with the requirements. Of these two, the forward approach probably fits better with respect to a conventional approach to correctness assessment.

In practice, using the TACO Traceability Model for assessment of functional correctness can be done in terms of the following steps.

1. For each requirement introduced, indicate whether it is a functional requirement. This can be done by means of mappings.
2. For each implementation of a requirement, indicate - by means of mappings - that it is an implementation.
3. For each functional requirement introduced, check that the forward traceability leads up to an implementation of this requirement. This can be done by:
4. For each functional requirement, check that the requirement is correctly implemented by validating the sequence of changes leading from the requirement to its implementation.

3.4 Requirements Understanding

One important aspect of the requirements understandability relates to the understanding of the interface between the application to be developed and its external environment (such as the physical plant). This requires that the software engineers understand the application domain and communicate well with the different stakeholders. To facilitate this communication, it might be necessary to specify the requirements in accordance with the different viewpoints the stakeholders have to the system, where each viewpoint provides a partial view of what the system is expected to provide. As a consequence, the requirements specification will cover different views on the same system, giving an additional dimension to the question of consistency between the different requirements. An important task of the software engineers is to integrate and reconcile the different views in such a way that contradictions are revealed and corrected.

In order to cope with the complexity of the resulting set of requirements, it is advisable to classify and document the requirements in accordance with the views they represent. This way of separating the concerns can provide a horizontal, modular structure to the requirements. Modularity provides several benefits in the requirements engineering process, including the capability to understand the system in terms of its pieces. This first of all relates to the fact that modularity allows separation of concerns, both with respect to the different views represented by the different stakeholders' expectations to the system and to different levels of abstraction. This makes it easier for the different stakeholders to verify their requirements, while at the same time providing a means for handling the complexity of the full set of requirements. The TACO Traceability Model can be adopted to facilitate this separation of concerns by relating requirements to the views they reflect. This can be utilized in different kinds of analysis of the requirements throughout the development of the system.

Some of the stakeholders may be unable to read the types of specifications preferred by the software engineers or mandated for the application. In such cases, the needs of the different stakeholders can be reconciled by providing (horizontal) traceability links between the, possibly formal, specifications used by the software engineers and more informal, natural language based expression of the same requirements. One could even consider providing links between the requirements and the user manual within the same traceability model. This could be utilized both for communication purposes and for the purpose of developing the user manual in parallel to the engineering of the requirements, which in some cases may be a recommended practice.

3.5 Implementation

The TACO Guidelines can be implemented in a variety of commercial or non-commercial tools extending the tools' capabilities by supporting relationship to diverse requirements sources in a formalized way and not only support the software development process from the specified requirements.

4 The TACO Network

The TACO project organisation is intended to constitute a Nordic expert network on requirements elicitation, specification, and assessment for digital I&C. The network provides a forum for exchanging experiences and research results on the questions to be addressed by the project, and provides a basis for evaluating the relative merits of the different practices, the relative importance of identified criteria, etc. A related concern is to facilitate knowledge transfer from other areas applying equipment that are used in NPPs.

The emphasis on best practices and identified success criteria means that the project needs to deal with real cases involving the development of a digital I&C system. By organising the project on basis of a Nordic expert network, the project contributes to the synthesis of knowledge and experiences, enhancement of competence on requirements elicitation, specification, and assessment, improved awareness of alternative practices, a basis for assessing current practices, and an incentive to search for best practice.

5 Further Work

The results from the TACO project will be utilized in the Nordic project MORE (Management of Requirements in NPP Modernization Projects, NKS project number NKS_R_2005_47). The overall objective of MORE is to improve the means for managing the large amounts of evolving requirements in NPP modernization projects. On basis of compiled experiences on the problem of handling large amounts of information in relation to Nordic modernization projects, the project will investigate how the approach to requirements management developed in the TACO project can be utilised to handle large amounts of evolving requirements in NPP modernization projects. While configuration management typically is file based, the TACO Traceability Model is paragraph based and therefore possibly more adequate for handling requirements (i.e., a requirement, or a composition of requirements, is treated as a single paragraph). The research will study how requirements can be grouped into concepts, and how design patterns can help to achieve this. One possibility is to utilise requirements (or design) templates, with guidance on how requirements can be decomposed or composed. The research will clarify how design patterns and requirements templates can be generated by utilizing the change history trees of the TACO Traceability Model.

References

[1] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques (Wiley, 1998).

[2] C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, 2nd edition (Prentice-Hall, 2003)

[3] P. Lindsay and O. Traynor. Supporting fine-grained traceability in software development environments. Technical Report No. 98-10, Software Verification Research Centre, School of Information Technology, The University of Queensland (July 1998).

[4] T. Sivertsen, R. Fredriksen, A. P-J Thunem, J-E. Holmberg, J. Valkonen, O. Ventä, J-O Andersson. Traceability and Communication of Requirements in Digital I&C Systems Development. Project Report 2004. NKS_R_2002_16. Nordic nuclear safety research (NKS, 2004).

An IEC 62061 Compliant Safety System Design Method for Machinery

Bengt Ljungquist and Thomas Thelin

Department of Communication Systems,
Lund University, Box 118, SE-221 00, Sweden
{bengtl, thomast}@telecom.lth.se

Abstract. The purpose of safety systems is to reduce dangers to human life or environment to acceptable levels. In order to aid companies in this when developing safety systems for functional safety of machinery, the standard IEC 62061 has recently been released. The standard proposes an outlined design method to follow requirements specification. However, companies that use the standard have to implement a design method on their own. This paper presents an implementation and enhancements to the design method in terms of using state machines and function block analysis documentation. The state machine connects the functional safety requirements with ordinary behaviour for equipment under control. The proposed method is evaluated in an industrial case and the main results from this indicate that the method works well, but needs tool support. Hence, the paper presents requirements for such a tool and discusses how it could be used to develop safety systems.

1 Introduction

Safety means absence from catastrophic consequences on the user(s) and the environment [1]. A safety-critical system is one by which the safety of equipment or a plant is assured. Examples of such systems are aeroplanes, nuclear plants and machinery systems. In order to develop safety-critical systems, structured methods are needed. Safety-critical development includes all project phases, from requirements specification with risk analysis, design, implementation, to verification and validation of the system. The development often involves both hardware and software (embedded systems).

This work is performed in a Swedish research project, SafeProd [8], which purpose is to develop guidelines aiding companies to interpret different safety standards. The paper describes application of the standard for functional safety of machinery IEC 62061 [2], which is based on the more general standard IEC 61508 [3]. IEC 62061 addresses both hardware and software safety issues.

There are three objectives of this paper:

– elaborate the design method for safety-related control functions in IEC 62061
– evaluate the proposed design method in an industrial case study
– define features for a tool that aim to help engineers to use the design method.

After the requirements have been specified and before the actual implementation of the system, the standard points out what should be done, but does not describe how. Hence, a company that implements the IEC 62061 is not aided on how to perform the design of the safety-related control functions. Additionally, the design method is very important in order to achieve the desired quality of the system. In this paper, we propose and clarify a design method to be applied when IEC 62061 is used in an industrial safety project. Furthermore, the method is evaluated in an industrial case and experiences from the evaluation are discussed. The main result of the evaluation is that the design method is appropriate to use and aid the implementation of safety-related control functions. However, in order to use it in a large scale, tool support is needed. No such tool exist today, but this paper outline some tool features.

The outline of the paper is as follows. In Section 2, the IEC 62061 standard for functional safety of machinery is described briefly including background and objectives, as well as basic terminology together with process and artefact requirements. Section 3 details the implementation of a design and development process according to the standard and also provides an example with a resulting artefact structure. Section 4 reports from an industrial case study in which the method was applied and executed. In Section 5 possible features of a tool supporting the design and development process are outlined. Section 6 then summarizes the paper with a discussion of results and further work.

2 The IEC 62061 Standard for Functional Safety of Machinery

2.1 Background

Automation together with requirements for larger production volumes and reduced physical effort have resulted in increased demands for Safety-Related Electrical Control Systems (SRECS) in order to achieve machine safety. Development of these SRECS employing complex electronic technology may be a difficult task requiring a considerable share of the total machinery development effort. As a response to these problems, IEC 62061 [2], a standard for safety of machinery and specifically functional safety of safety-related electrical, electronic and programmable electronic control systems, has been developed.

2.2 Objectives and Scope

IEC 62061 is a standard specific for machinery within the framework of IEC 61508 [3] and it was published in January 2005. It intends to facilitate the performance specification of safety-related electrical control systems in relation to the significant hazards of machines. Furthermore, it defines an approach and provides requirements to achieve the necessary performance of SRECS. The standard is intended to be used by machinery designers, control system manufacturers, integrators and others involved in the specification, design and validation of a

SRECS. The approach and requirements address assigning safety integrity levels, enabling SRECS design, integration of safety-related subsystems and SRECS validation. Requirements are also provided for information for safe use of SRECS of machines that can also be relevant to later SRECS life phases.

IEC 62061 provides safety by means of employing SRECS as parts of safety measures that have been provided to achieve risk reduction and thereby avoiding faults [1]. Additionally, the electrical control system that is used to achieve correct operation of the machine process contributes to safety by mitigating risks associated with hazards arising directly from control system failures, i.e., tolerating faults [1].

2.3 Basic IEC 62061 Terminology [2]

IEC 62061 has a detailed list of terminology from which we here present only the most essential for understanding this paper in Table 1.

Table 1. IEC 62061 Terms

Term	Description
Safety-Related Electrical Control System (SRECS)	Electrical control system of a machine whose failure can result in an immediate increase of the risk(s)
Safety-Related Control Function (SRCF)	Control function with a specified integrity level that is intended to maintain the safe condition of the machine or prevent an immediate increase of the risk(s)
Subsystem	Entity of the top-level architectural design of the SRECS where a failure of any subsystem will result in a failure of an SRCF
Function block	Smallest element of a SRCF whose failure can result in a failure of the SRCF. Note that this definition is not equivalent to the function block as a programmatic concept described in IEC 61131-3 [11]
Electrical control system	All the electrical, electronic and programmable electronic parts of the machine control system used to provide, for example, operational control, monitoring, interlocking, communications, protection and SRCFs
Safety Integrity Level (SIL)	Discrete level (one out of a possible three) for specifying the safety integrity requirements of the SRCFs to be allocated to the SRECS, where SIL three has the highest level of safety integrity and SIL one has the lowest. Note that this definition differs from IEC 61508

2.4 Standard Process Requirements

In order to support SRECS design and development, the IEC 62061 standard imposes a structured development method to be used for design and architecture elicitation. However, it does not provide detailed guidance of the composition of such a method but outlines it to include ten steps and provides detailed requirements for which information to be produced. It is up to the user of the standard to implement a method fulfilling these requirements. The first steps are as follows according to the standard [2]:

1. Identify the proposed SRECS for each SRCF from the safety requirements specification (SRS)
2. For each function decompose the SRCF into function blocks and create an initial concept for an architecture of the SRECS
3. Detail the safety requirements of each function block
4. Allocate the function blocks to SRECS subsystems
5. Verification

This paper focuses on these steps since these, this far, have been the scope of the research project (see Chapter 4) in which this paper has been produced. However, the standard also outlines a five-step procedure following these steps concerning subsystem realisation and design.

2.5 Artefact Structure Requirements

IEC 62061 requires that the SRS shall describe the functional safety and safety integrity requirements of each SRCF to be performed. This includes the conditions of the machine in which the SRCF shall be active. The functional safety requirements of the SRCF shall then be further decomposed into so called function blocks forming a logical AND of the SRCF. A function block is defined as the smallest element of an SRCF whose failure can result in a failure of the SRCF. They are thus an abstraction of the functional components and should not be confused with its physical structure. The standard then requires that the function blocks structure should be documented in terms of:

– A description of the structure
– The safety requirements for each function block
– Inputs and outputs and internal logic of each function block

Furthermore, the standard requires that function blocks shall be allocated to subsystems, realizing functional safety requirements. Non-functional safety requirements like maintainability and integrity [1] are then captured in the description of the subsystems. If the safety requirements of the subsystem cannot be realized with a single component, then the subsystems must be further decomposed into subsystem elements. When this structure is clear at an element level, the function blocks are decomposed into function block elements. These are then mapped onto subsystem elements in a many-to-one relationship with many function block elements with similar functionality in different SRCFs being allocated to a single subsystem element. For example, motion sensing in different SRCFs is mapped to an inductive motion sensor subsystem. In Fig. 1, an example artefact structure is provided. For simplicity, the function blocks and subsystems consisting of only one element have not been illustrated as being composed of single elements. Hence, the standard provides a structured design framework in which each part of the safety related electrical control system is traceable from hazard analysis to physical components of the SRECS.

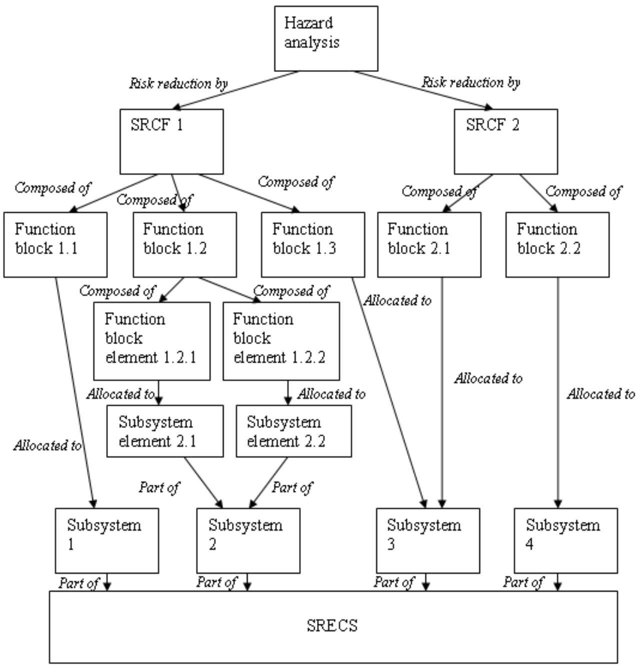


Fig. 1. Example of a design artefact structure according to the IEC 62061 standard

3 Design and Development Method Implementation

3.1 Process

In Fig. 2, a suggested process implementing the requirements stated in Section 2.4 is shown. This process follows the same basic outline as prescribed by the standard and have been used in the research project. Below follows step-by-step details on how these requirements may be implemented. Also, an overview which artefacts that are related to which documents is provided in Fig. 3.

Step 1 - Refine SRS. The system analyst revises the SRS specified by the requirements specifier. He/She relates the SRCFs to the behaviour of the EUC (Equipment Under Control) by using a state machine denoting in which states of the normal equipment behaviour that the SRCF is active. The IEC 62061 does not prescribe the SRS to include such a state machine, but it proved important for method success as described in Section 3.2.

Step input: FBSD

Step output: Refined SRS

Steps 2-3 - Function block analysis and architecture prototyping. The architecture and design phase starts with the system analyst breaking down the functional safety requirements into function blocks and documenting them as required. In this step the system analyst should communicate with system developers implementing a prototype according to the suggested function block

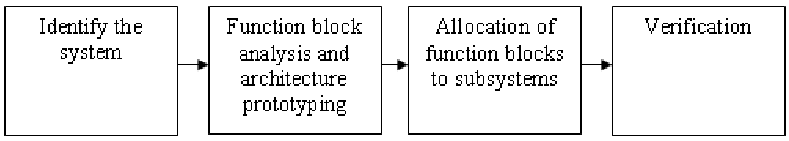

Identify the system
Function block analysis and architecture prototyping
Allocation of function blocks to subsystems
Verification

Fig. 2. The implemented design process

structure. By this early prototype implementation, it is possible to validate that the suggested architecture is feasible to implement. The function blocks structure shall be captured in a function block structure document. In this document, it shall be specified to which SRCF that each function block is traced. This facilitates verification since it is possible to trace a safety requirement from specification to implementation. Furthermore, the function blocks inherit the SIL of the SRCF they are traced to.

Step input: Refined SRS

Step output: Function Block Structure Document (FBSD)

Step 4 - Allocation of function blocks to subsystems. Finally, the system analyst identifies, in terms of physical components, the subsystems needed to implement allocated function blocks processing their inputs and outputs. Typically, a subsystem implements many function blocks with similar functionality. For example logic function blocks of the different safety related control functions are likely implemented by the same PLC subsystem. This is documented in a subsystem definition document. For this step it is enough to name the different subsystems since these will be specified and documented in detail in later steps (that is steps 6-10).

Step input: FBSD

Step output: Outlined subsystem definition document (SDD) with subsystems named and traced to function blocks.

Step 5 - Verification. A review meeting then takes place together with the safety requirements specifier and system developers in order to verify safety requirements and to ensure that the suggested solution is possible to implement. If the review meeting does not approve the suggested solution, then steps 2-4 are iterated until the solution is acceptable and approved in a new review meeting.

Step input: Refined SRS, FBSD, Outlined SDD

Step output: Verified SRS, verified FBSD,verified SDD.

3.2 Example SRECS Design - Rotation Speed Supervision

In order to further illustrate the method, a short example is provided below. This example is partly fictional and partly using some safety functionality as discovered during the industrial case study described in Section 4.

In the example the EUC, a machine in an automation environment, has a rotating part normally moving at high speed. In order to specify its behaviour, the EUC has been implemented according to the ANSI/ISA S88 standard state transition model [9]. However, the machine is currently running without any

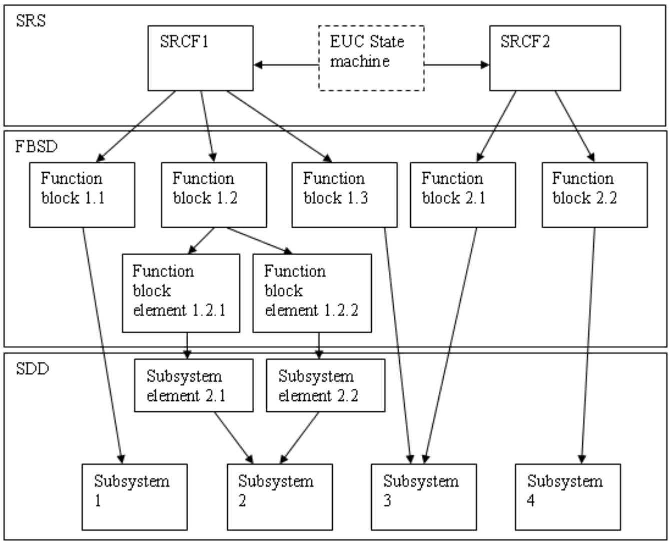


Fig. 3. Artefact-to-document mapping in method implementation

protections. If operators are caught by this part while it is moving, result may be severe injuries. There is a need to reduce the risk for this during normal operation. However, there is also a need to maintain and clean the machine with power enabled in order to shorten machine down times. In addition to a safe stand still under these conditions, there is a need for the rotating part to move at low speed during maintenance and cleaning in order to access the entire part easily. For the purpose of reducing risks to acceptable levels, three different SRCFs are used:

– Guard Door - active during normal machine operation. If guard door is not closed, it is not possible to empower the EUC, except for when one of the other two safety functions is active.
– Safe Stand Still - active during cleaning and maintenance. Rotating part must not move, but power to electrical drives is enabled.
– Safe Low Speed - active during cleaning and maintenance. Rotating part speed must be less than a specific value if low speed is requested.

In accordance with Step 1 of the method described above, a state machine is then elicited for the example, see Fig. 4.

To improve diagram clarity, the transient states (states specifying the machine is under way to another state) that the ANSI/ISA S88 standard suggests for machine behaviour have not been accounted for in Fig. 4 above. The state machine thus relates the safety functions to the ordinary EUC behaviour. In the diagram it is indicated that the SRCF guard door is active during the states HELD and RUNNING and then leaves the task of keeping the system safe to the SRCFs Safe Stand Still and Safe Low Speed if the guard door is opened.

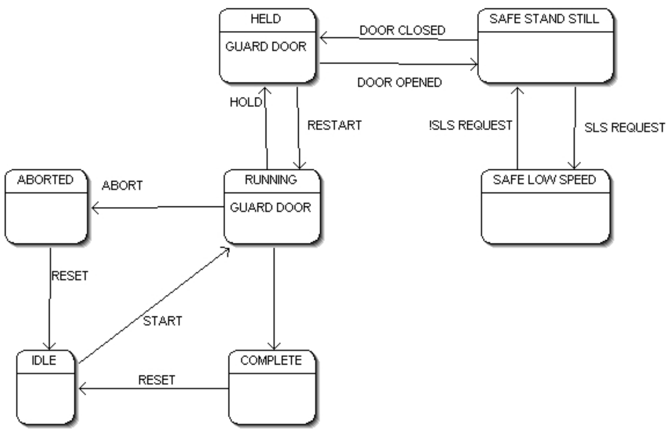


Fig. 4. State machine diagram example relating EUC behaviour to SRCFs

It then follows from steps 2-3 of the method that the SRCFs should be broken down into function blocks specifying input and output relationships. The guard door SRCF is broken down into function blocks door sensing, logic and power switching. Furthermore, the Safe Stand Still and the Safe Low Speed SRCFs are then broken down into function blocks speed sensing, logic and power switching. The next step, allocation of these function blocks to subsystems (step 4), is then done as follows:

– Door sensing → Electromechanical door position sensor
– Speed sensing → Encoder signal sensor
– Power switching → Contactor and server drive

In this example neither details of each function block, function block diagrams nor subsystems details are provided due to the limited space. We do though provide an artefact overview in order to illustrate the artefact relationship, see Fig. 5. In this figure we also see that the door position sensor is duplicated. This illustrates the concept of duplicating sensors due to IEC 62061 requirements on number of tolerated hardware failures on a subsystem for a specific safe failure fraction [2] and required SIL.

4 Industrial Evaluation

4.1 Project Characteristics

The standard and the suggested enhancements were applied in a currently running research project, SafeProd [8]. The project includes the following participants:

– Software experts and system analysts - Lund University
– Hardware experts - Swedish National Testing and Research Institute
– Machinery manufacturer where EUC development project is sited

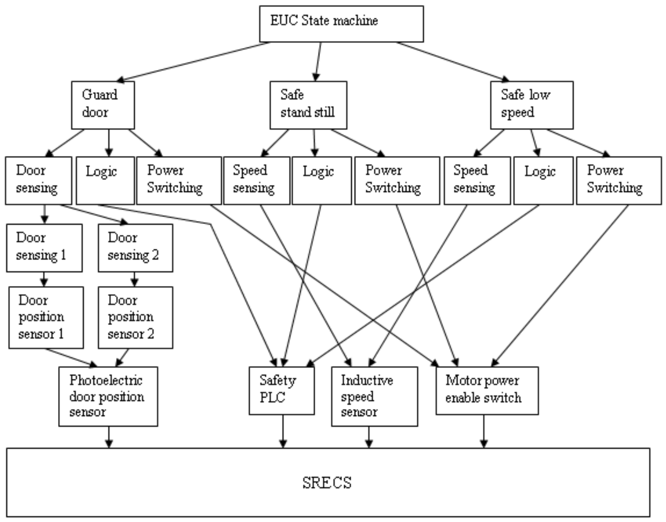


Fig. 5. Resulting example artefacts relationship

The objectives of SafeProd are to apply the IEC 62061, IEC 61508 and IEC 61511 (process industry sector specific safety standard) standards to industrial problems in three separate sub projects and publish guidelines which aim at faciliting the application of these standards in Swedish industry.

4.2 Method Execution

The SRS was created according to a template, which has been elicited in the project using MS Word. Initially there was no state machine present in the SRS produced. This had the effect that the system analyst did not know when the SRCFs was supposed to be active, since the requirements did not address this. In the light of this hindsight, a state machine model of the EUC was elicited using the tool Omondo EclipseUML Studio [10] and UML [13] state diagrams in which the SRCFs were denoted as either new states or actions during transitions between states. These diagrams were then added to the specification. A function block analysis was then made of the SRS, resulting in a function block structure document written in MS Word in accordance with the method. In this document, information according to the IEC 62061 requirements for function blocks (see Section 2.5) was made available.

The document was elaborated iteratively using project meetings for approval of the documentation that the system analyst had elicited. The function blocks with similar functionality in different SRCFs were then allocated to common subsystems (e.g. sensors sensing the same type of data, like torque sensors and safety PLCs realizing logic function blocks). Implementation of an architectural proof of concept according to this function block structure is currently just about to begin in the project. Also, the project currently is documenting these subsystems into a subsystem definition document written in MS Word, in which requirements according to the standard are collected for each of the subsystems.

4.3 Experiences

The system analyst, who performed most of the design analysis, experienced that the state diagram notation facilitated communication of the ordinary behaviour of the EUC state model, based on that of the ANSI/ISA S88 standard [9] and assisted understanding and communication when the SRCFs should be active. The system analyst was unable to functionally decompose the SRCFs to function blocks until state diagrams were available.

The function blocks analysis allowed the SRECS functional requirements to be specified precisely and correctly. The ambiguities related to what information that was to be recorded was dealt with and the input and output to the SRCFs were identified. Furthermore, specifying the functional requirements of the SRECS in function blocks and the non-functional (except for safety requirements which were stated at a function block level) requirements at subsystem level proved to be an effective way of structuring the system requirements. This had previously been experienced as a problem by the industrial project. The functional requirements are typically derived from scenarios of SRCF behaviour and the non-functional requirements are more likely derived from expected hardware behaviour. Separation of these into an abstract functional layer and a physical non-functional layer thus benefited SRECS development greatly.

In addition to the system analyst experiences, separate follow-up interviews with a project leader as well as a advisor for product safety of the machinery manufacturer organisation were conducted separately. These interviews provided the following information:

- Regarding formerly used design methods, these have previously been of lesser complexity. Risk analysis has provided sufficient information for direct implementation. Furthermore, safety systems have been mostly in form of COTS safety components. There is however increasing needs of customising safety functions, which may not be offered by these.
- When it comes to the terms of expectations upon the standard before the project, the respondents expected more sophisticated safety functions allowing increased production time with adequate safety. Previous standards like EN954-1 [12] have provided more primitive safety functions like powerless machine states, disturbing production flow when active. This has sometimes made users deliberately by-passing safety functions in order to increase production. There are also expectations of being able to handle complexity and modularisation better; by component and block structures in safety, equipment components that have been analysed individually may be put together with a minimum of effort for safety integration testing. Both respondents felt that it was too early in the project to consider the standard as a success regarding meeting these expectations, but the result this far was felt to be promising.

– Concerning the techniques used, the state machine diagramming approach felt natural to use when describing the ordinary EUC behaviour and the extension of this using SRCFs. In development of the EUC ordinary control system using a state machine together with naming conventions have been key success factors and it seems natural to use these also for the derivation of the SRECS.
– The function block analysis technique appeared to clarify the structure of the SRECS and it was concluded that the function block structure document may be used for verification purposes checking that no "extra" functions have been added, which sometimes have been the case in some previous system development projects. However, one of the respondents meant that the function block analysis seemed like a little bit overhead effort, and would like to get to realisation quicker.
– The concept of requirements layering, that is, mapping functional requirements to function blocks and non-functional requirements to subsystems, was also perceived as natural. The respondents felt as if non-functional requirements must be solved at physical level. The structure is in alignment with and may support the iterative hazard analysis that currently is carried out in the organisation with an up-front analysis finding hazards at a system level and then detailed analysis at a component level finding failures related to components and applications.

In addition to the findings described above, it was discovered that there was a need for a tool supporting the design development process. This is mainly motivated by the following problems encountered during the project:

– Automated development tasks provide shorter SRECS development and documentation time and thus implies cheaper projects and provides argument for justification of implementing safety even with a limited budget.
– Automated development tasks provide better quality and thus better safety. Examples of such tasks are artefact traceability management, state machine documentation and IEC 62061 standard compliance checks.
– In order to be able to verify requirements and manage change reports (e.g. incident reports indicating flaws in SRECS implementation) traceability management is needed. Traceability is easily corrupted if performed manually, e.g. an update of a document in one section must be followed up by updating dependent sections in other documents.
– State machine modelling is best done graphically. Capturing graphical models is best done with a tool.
– Communication problems. A common model accessible for all project members helps settling disagreement occurring due to knowledge being in people's heads and not in the organisation. This could be achieved by a tool.
– Problem with grasping the entire standard and remembering to use standard requirements in the development process at the right time. Knowledge should be in the tool and not required to be in peoples heads [4]. The organisation using the IEC 62061 in a development project would then have a much flatter

learning-curve. Using similar arguments, an example of a tool facilitating implementation of IEC 61508 have been provided by Faller [5].

– IEC 62061 requires documentation which has to be captured in some way. A tool (more advanced than a word processor) could provide a structure for this. If using a tool not more advanced than a word processor, it is much up to the analyst to maintain the document structure.

5 Design Process Tool Support

As mentioned and motivated in previous section, there is a need for a tool facilitating the development process outlined in Section 3. Implementing such a tool is out of scope of this paper, but we do provide a feature overview below, including a screen shot from a tool prototype.

– A tool that is used for supporting standards must be able to change as standards change. Therefore, it should support changing development process requirements reflecting different versions of standards by taking process definition as input. This could be done in suitable XML-format[14] like SPEM[7].
– Regarding documentation, the tool should support making state machine diagram modelling SRCFs to be active during either states (either in existent EUC states or being states on their own) or in transitions. This feature is shown in the screen shot in Fig. 6.
– Function block and subsystem documentation (that is the documents FBSD and SDD) could be in form of a requirements database helping the user by requiring specific input data for each artefact.
– The tool should support collaborative and distributed development.

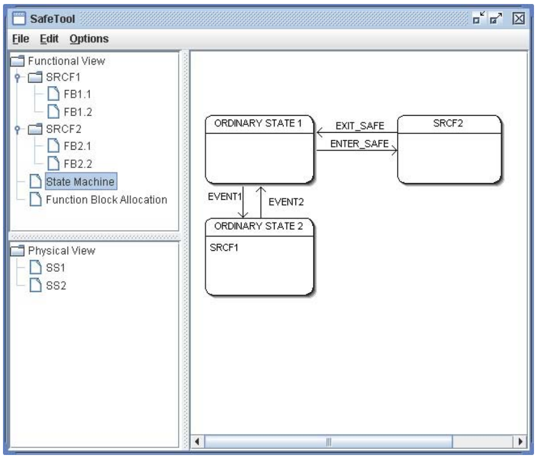


Fig. 6. Prototype screenshot of State machine diagram feature

- To support reporting to management and handouts of system models to be discussed at meetings, the tool should have documentation export functionality to proper document formats.
- To reduce the burden of the engineer of having the whole standard in mind when developing systems, the tool should have development process decision support (i.e. "what to do next"). This can be used for driving the project in terms of feedback e.g. on not yet produced artefacts.
- One experience during the research project was that graphical models of the artefact model facilitated the process of understanding it. Therefore, the tool should support graphical function block structure modelling, in order to visualize function block input-output relationships. Another process that could be supported by graphical modelling is traceability management with graphical function block-to-subsystem mapping.
- The tool should have querying capabilities to support finding the right information in a large system model, like finding functional requirements of subsystems from assigned function blocks.

6 Discussion

Where previously method requirements of standards have been poor or not existing at all, the IEC 62061 standard shows a possible way of filling this gap by stating requirements and providing guidelines for the system design process of a SRECS. This paper has taken the method of this standard further and provided a more hands-on method for implementation of a design process according to the standard. It defines a step-by-step method which could be used in a SRECS development project for functional safety of machinery and clearly defines the input and output documentation of each step, which the standard does not.

The state machine approach of modelling when SRCFs should be active turned out to integrate well with the existent way of specifying EUC behaviour by extending already existent specifications in form of state machines. It was perceived as clarifying by project members when the SRCFs should be active, which was hard to define in natural language, even if this technique was appropriate according to the standard. Furthermore, the function block analysis facilitated structuring of information by requirements layering as prescribed by the standard. Performing this had previously been perceived as a problem by the machinery manufacturer. However, some project members perceived it as overhead analysis.

However, it remains to show in the future whether the method prescribed by the standard at large is valid. The focus of this paper is to report from the design method. Furthermore, the general validity of the design method suggested by this paper should be studied in other projects independent from this.

The paper presents some basic features of a tool, which currently is being developed in the research project. Furthermore, case studies regarding the effects of using the tool in development projects should be performed. One of the major research questions of these studies will be demonstrating the feasibility of the concept of transforming standards to guiding tools.

It has also been indicated in the project that the suggested method could be improved. One thing that would be interesting to model is SRFCs overriding each other by modelling them as a hierarchy of state machines. For example, an emergency stop would override all other functions of a SRECS. This cannot be captured by a single level state machine diagram. A similar technique of applying hierarchies of state machine is shown by Papadopoulos [6], but has there been applied for purposes of safety monitoring, whereas the improvements of the model suggested in this paper would be regarding hierarchies of safety functions.

Acknowledgements

This work was partly funded by The Swedish National Agency for Innovation Systems (VINNOVA). The authors would like to thank Per Runeson and Karl-Erik Årzen for their valuable ideas and comments regarding this paper. We also would like to thank all SafeProd project members for their helpful input.

References

[1] Laprie, J-C., Dependability: basic concepts and terminology, dependable computing and fault-tolerant systems. Springer Verlag, Wien-New York, 1992
[2] IEC 62061: Safety of machinery Functional safety of safety-related electrical, electronic and programmable electronic control system. International Electrotechnical Commission, 2005
[3] IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, parts 1 to 7, International Electrotechnical Commission, 1998 and 2000
[4] Norman, D., The Design of Everyday Things, Doubleday/Currency, New York, 1988
[5] Faller, R., Project experience with IEC 61508 and its consequences, Safety Science, Volume 42, Issue 5, 2004, pp 405-422
[6] Papadopoulos, Y., Model-based system monitoring and diagnosis of failures using statecharts and fault trees, Reliability Engineering & System Safety, Volume 81, Issue 3, 2003, pp 325-341
[7] Software Process Engineering Metamodel Specification, Version 1.1, Object Management Group, 2005
[8] SafeProd, http://www.sp.se/safeprod
[9] ANSI/ISA 88.01-1995, Batch Control, Part 1 : Models and Terminology.
[10] EclipseUML Studio, Version 1.1.0, Omondo, 2004
[11] IEC 61131-3: Programmable Controllers, Part 3: Programming Languages, International Electrotechnical Commission, 1993
[12] EN-954-1: Safety of Machinery Safety related parts of control systems, Part 1: General principles for design, 1996
[13] The Unified Modelling Language Specification, Version 1.5, Object Management Group, 2003
[14] Extensible Markup Language (XML), Version 1.1, W3C Recommendation, 04 February 2004

Design Evaluation: Estimating Multiple Critical Performance and Cost Impacts of Designs

Tom Gilb

Tom@Gilb.com

Abstract. How should we evaluate someone's design suggestion? Is gut feel and experience enough for most cases? Is anything more substantial and systematic possible? This paper outlines a process for design evaluation, which assesses the impacts of designs towards meeting *quantified* requirements. The design evaluation process is viewed as consisting of a series of design filters.

1 Introduction

This paper proposes that: (1) Design evaluation is primarily a matter of understanding the 'performance and cost' impacts of the design(s) numerically, in relation to quantified performance and cost requirements; and (2) Design evaluation needs to go through several maturity stages, but remains fundamentally the same question: *what does the design contribute to meeting our requirements?*

Prior to design evaluation, the requirements must be specified. All the performance and cost requirements must be specified quantitatively. The requirements should be subject to specification quality control (SQC). An entry condition into the design evaluation process should be that the requirement specification has successfully exited quality control with an acceptable level of remaining defects per logical page (for example, less than one remaining defect per 300 words). The design evaluation process consists of several maturity stages, which can be viewed as design filters. These maturity stages include:

- **Value-based selection:** Select the requirements with the highest stakeholder value;
- **Constraint-based elimination:** Delete designs that violate constraints;
- **Performance-based selection:** Pick the most effective remaining designs;
- **Resource-based optimization:** Select the effective designs that are most efficient – effect/cost;
- **Risk-based elimination:** Evaluate designs based on performance and cost risks.

Each of these evaluation filters requires specification and estimation tools that are common sense, but are not commonly taught or employed. These tools are based on the defined Planning Language ('Planguage'), developed by the author for practical industrial use through many years [1]. Let's now go through the design evaluation process in greater detail. The main points will be summarized as a set of principles.

2 Clear Specification of Requirements

Principle 1: A design can *only* be evaluated with respect to *specific* clear requirements.

The foundation of design evaluation is a set of clear requirements. Any evaluation of any design to try to ensure meeting vague requirements is going to be imprecise[1]. If you look around you, both inside the systems engineering community and outside it, you will observe that people commonly evaluate designs 'in general terms' (for example, "*that* design is the *most* user-friendly…"), rather than with respect to specific *immediate*, and *residual*[2] project or product requirements.

One of several reasons for this generalizing about 'good designs,' is that we are very vague with requirements specification. Our most critical requirements are typically unclear, and not quantified. In my consulting and teaching practice I see this happening worldwide. When I evaluate such requirements using SQC, there are invariably approximately 100 defects present per page, unless special effort has been made to eliminate them [2]. To the degree that is the case, we cannot readily expect anyone to perform a logical evaluation of the suitability of a given design for a fuzzy requirement. Consider the following questions: How good is a 'Mac interface' for getting 'higher usability'? And how good is 'MAC OS X' compared to 'Windows' for 'higher security'?

These are silly questions because the requirements are not clearly defined (Note also that the other security designs that will co-exist are not listed and analyzed). Now, requirements are not the primary subject of *this* paper[3]. So we need to be brief on them, so as to concentrate on design itself.

But here are some of the things I would insist are necessary pre-requisites for being able to evaluate a design:

- All performance (including all qualities) and cost requirements are expressed quantitatively (with defined scales of measure). Not just nice sounding words.

  ```
  Scale: Minutes for defined [Tasks] done by defined [People].
  ```

- All performance requirements must include at least one *target* (A target is a level we aim for) and at least one *constraint* (A constraint is a level we aim to avoid with our design).

  ```
  Goal: 3 minutes.
  Fail: 10 minutes.
  ```

[1] This is not the same as demanding that the requirements are known upfront: requirements should not be 'frozen' and they should be allowed to evolve over time. The issue here is that the known or predicted requirements are expressed clearly.

[2] Residual requirements: Residual: Concept *359: The remaining distance to a target level from a benchmark or current level (From Planguage Glossary in [1]). The point being that design is a sequential process of evaluating necessary designs, to add onto the current set of designs – and the only designs necessary at any point in the process, are designs that will move us from the performance levels we estimate we have reached, with the current set of designs, towards our required Goal levels of performance. A good analogy is the 'next chess move'.

[3] See [2] for further discussion on requirements.

- All performance requirements must include explicit and detailed information regarding the *short-term* and the *long-term* timescales of expectation.

```
Goal [Release 1.0]: 3 minutes.
Fail [Release 1.0]: 10 minutes.
```

- All relevant *constraints* on solving the design problem are specified complete, officially, explicitly, unambiguously, and clearly. This includes all notions of restrictions such as legal, policy, and cultural constraints. It also includes any known design constraints (such as from our own architecture specification). Constraints will consider all necessary aspects of development, operations, and decommissioning resources.

Clear and complete requirements are a set of basic entry conditions to any design or architecture process. Without it a design process is like a fighter plane with no known enemy, like a passenger ship at sea with no destination port identified, or like a great invention with no market. Design evaluation is quite simply about deciding how well a design meets the total set of requirements.

Principle 2: All designs have performance and cost attributes, but not necessarily the ones *you* require.

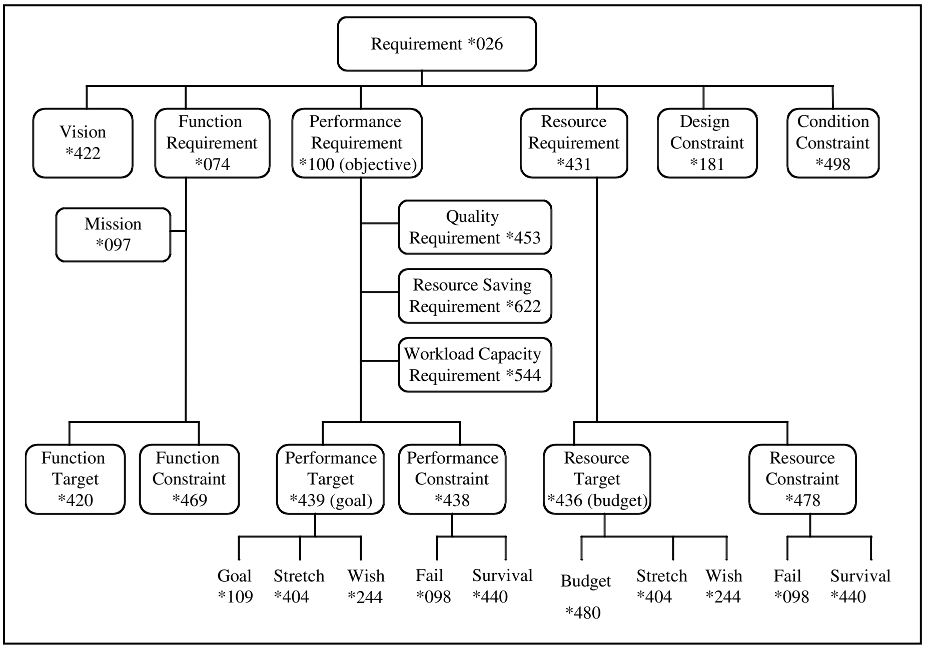

Requirement *026
Vision *422
Function Requirement *074
Performance Requirement *100 (objective)
Resource Requirement *431
Design Constraint *181
Condition Constraint *498
Mission *097
Quality Requirement *453
Resource Saving Requirement *622
Workload Capacity Requirement *544
Function Target *420
Function Constraint *469
Performance Target *439 (goal)
Performance Constraint *438
Resource Target *436 (budget)
Resource Constraint *478
Goal *109
Stretch *404
Wish *244
Fail *098
Survival *440
Budget *480
Stretch *404
Wish *244
Fail *098
Survival *440

Fig. 1. A map of the requirements concepts, which includes a variety of constraints. The *nnn are references to detailed definitions of these concepts in the Planguage Glossary [1].

3 Value-Based Selection

Principle 3: The real value of a design to a stakeholder depends partly on the technical characteristics of the design, and partly on the planned, perceived and actual use of those characteristics in practice, over time.

The value of a design depends on the stakeholder view taken. The producer of a product has one view. The users of a product have another view. It is going to be the producer of a product who will directly and primarily evaluate designs from *their* point of view. They ideally will try to maximize their profitability or service delivery. The commercial producers will do this by maximizing the value delivered to their customers, so that their customers will 'return the favor' by paying well, in terms of price and volume. It should be possible to evaluate a design market, segment by market segment, for estimated sales or profit as a result of it. Ideally your marketing people would make such an evaluation. The *service* providers (such as military, space, government) will worry about value to their stakeholders for money spent.

<Name tag of the performance requirement or cost requirement>:

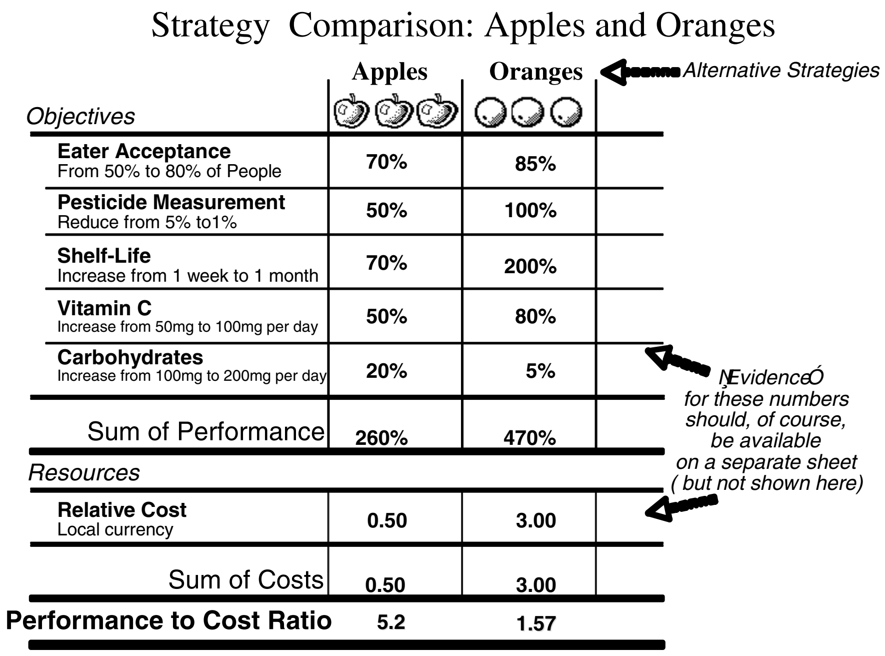
Ambition: <Give overall real ambition level in 5-20 words>.
Version: <Each requirement specification should have a version, at least a date, yymmdd>.
Owner: <The person or instance allowed to make official changes to this requirement>.
Type: <Performance|Cost Requirement>.
Stakeholder: { , , }. "Who can influence your profit, success or failure?"
Scale: <Defined units of measure, with [parameters] if you like>.
Meter [<qualify which version and level>]: <Specify how you will measure>.
==== Benchmarks ============ the Past ==================================
Past [<time>, <place>, <event>]: <Actual or estimate of past level> <- <Source of past data>.
Record [<time>, <place>, <event>]: <Actual or estimate of record level>] <- <Source of record data>.
Trend [<time>, <place>, <event>]: <Prediction of level> <- <Source of prediction>.
==== Targets ============= Future Needs ===================================
Wish [<time>, <place>, <event>]: <- <Source of wish>.

For Performance Requirements Only – Use 'Goal'
Goal [<time>, <place>, <event>]: <Target level> <- <Source of goal>.
Value [<stakeholder>]: <Refer to what this impacts or how much it creates of value>.

For Cost Requirements Only – Use 'Budget'
Budget [<time>, <place>, <event>]: <Target level> <- <Source of budget>.
Stretch [<time>, <place>, <event>]: <Motivating target level> <- <Source of stretch>.
==== Constraints ==
Fail [<time>, <place>, <event>]: <- <Source of fail>. "Failure Point"
Survival [<time>, <place>, <event>]: <- <Source of survival>. "Survival Point"

Fig. 2. This figure shows a Planguage requirement template with hints. This gives some idea of the basic parameters that should be used to describe a performance or cost requirement quantitatively [1].

Table 1. A symbolic example of evaluating two different 'designs' for 'which fruit to buy'. This is a simple Impact Estimation table application. The % estimated impact of a design is on a scale where 100% means the design brings us to the Goal level on time. 0% means there is no impact compared to some defined benchmark level, such as the previous system state.



Strategy Comparison: Apples and Oranges

Objectives	Apples	Oranges	
Eater Acceptance From 50% to 80% of People	70%	85%	
Pesticide Measurement Reduce from 5% to1%	50%	100%	
Shelf-Life Increase from 1 week to 1 month	70%	200%	
Vitamin C Increase from 50mg to 100mg per day	50%	80%	
Carbohydrates Increase from 100mg to 200mg per day	20%	5%	
Sum of Performance	260%	470%	
Resources			
Relative Cost Local currency	0.50	3.00	
Sum of Costs	0.50	3.00	
Performance to Cost Ratio	5.2	1.57	

Alternative Strategies
ÒEvidenceÓ for these numbers should, of course, be available on a separate sheet (but not shown here)

If the marketing people are not involved or helpful, the technologist is left to look at the contribution of a given design to the *performance requirement* levels. Designs have value primarily as long as they help us move to the Goal levels, and perhaps to the degree they help us move to the Stretch levels (see Figure 2). Beyond those target levels, a design does not have any formally agreed value, because it is not formally required.

So we need to find the designs that satisfy the prioritized agreed target levels, at the lowest costs and risks available. I have developed an Impact Estimation (IE) method to help us see the contribution of design ideas to the requirement levels, the degree of risk involved and the corresponding development costs (See later, Table 1). So we can make a rational decision and present it to others.

4 Constraint-Based Elimination

Principle 4: It doesn't matter how good or how cheap a design is, if constraints forbid it.

We are assuming that there is a flow of one or more design ideas to be evaluated. The question of how we identify these design ideas is a separate topic. Before we go deeper into the design, we need to assess if any design idea is disqualified by any requirement.

We need to pass the design through the set of design filters known as constraints. The questions to be asked include:

- Does the design violate any specified design constraint?
- Does the design violate any condition constraint?
- Does the design violate any performance constraint or cost constraint?
- Does the design in *combination* with other design elements, adopted or projected, threaten to violate any constraint?

Because if a design violates, or threatens to violate, any defined constraint, a design needs to be set aside in favor of designs that do not. Later we could, if necessary, discuss relaxing a constraint, or risking or tolerating a constraint violation, in order to make use of an otherwise superior design, so we need to be careful about *permanently* discarding designs that initially violate some constraint. They might turn out to be the best design of all. So, 'set aside', preferably with *annotation about the constraint violation*.

```
Design X: <Detailed description>.
Status: Set Aside <- Tom, November 13, 2004.

Rationale: Threatens to violate cost constraints as it alone
takes 90% of the budget.
```

I seriously suggest that all rejected designs be *formally kept* in the systems engineering documentation, with their status and rationale for rejection.

Principle 5: Designs should not be rejected permanently. The reasons for rejection should be clearly documented, the design specification kept; and the rejection possibly re-evaluated later.

5 Performance-Based Selection

For the set of proposed designs that survive constraint evaluation, the next step is to evaluate which ones have the *best set of impacts* on our *required performance target levels.*

Principle 6: The major capability of a design is its ability to contribute to required residual performance levels.

We fail to evaluate designs in all critical dimensions. I find that systems and software engineers, in too many cases, do not even do a systematic evaluation of a design along a single performance dimension (such as 'Reliability'). But, even if they did do that, there is another evaluation problem to confront. Designs have potential impacts on many of our most critical performance requirement dimensions. Real systems seem to have about 20 to 40 performance dimensions that people are willing to set quantified requirements for, and to evaluate. One dimension is not enough. We need to look at:

- Other major secondary contributions to critical requirements;
- Possible negative side effects on the critical requirements.

We usually do not have good enough facts about the design impacts. Anything less than a thorough examination of the potential impacts of a design in *all* critical performance requirement dimensions, is irresponsible design engineering. The major initial outcome of any systematic quantitative evaluation in these many dimensions is, initially, 'shocking'.

It turns out that even our most expert designers do not even *claim* to have any *factual* knowledge about *most* of the performance impacts of a design specification, in all our specified requirement dimensions! This may seem hopeless: 'Knowing that we do *not* know'. But in a sense it is the beginning of wisdom, and there is a systematic approach to dealing with this ignorance – that is the subject of this paper. But we would do well to recognize this ignorance initially, clearly, and publicly, in our design engineering processes. Recognize the initial level of knowledge about a design, and then act cautiously as we progress the design towards serious commitment.

What is the alternative to a systematic initial design impact evaluation process? We do not have to 'act like engineers' and evaluate designs in a systematic and quantitative way. We can just 'decide to implement them and see what happens'. The problem there is that it may be too late to use better designs, and it would perhaps have paid off to do more engineering evaluation earlier.

There are interesting options between the extremes of 'full ignorance/high risk', and 'expecting perfect research data for all impacts of all design candidates'. For example evolutionary methods [1], [4], [5], [6] may allow us to remove some of our ignorance about a design, at relatively low risk (By designing and implementing small Evo steps, we can ensure the maximum potential project loss 2% for a design that is a total failure). The fact that we rarely have the facts we need, in order to evaluate designs properly, is not a good reason to avoid trying to evaluate them quantitatively, before final commitment to using them. The lack of facts is a warning signal about risks. It can lead directly to more realistic expectations. It can also lead to risk mitigation tactics in contracting, alternative conservative design specifications, or lead to doing experimental steps to get needed data before scaling up – all traditional good engineering tactics.

Principle 7: A design will be best understood in terms of its multiple quantified impacts on your residual requirements.

How far should we go in evaluating a design? It is not enough, in my opinion to let your in-house expert loose, to make estimates of a design's impact on performance levels. They should be asked (in your systems engineering standards!) to document the basis for the estimates, and the basis for the uncertainty of their estimates. An example of doing this is given in Table 2 using the Impact Estimation method.

Notice that for each estimate we ask for the *uncertainty boundaries* (worst case/best case). We ask for *evidence* –the facts backing the estimate. We ask for the *source* of the evidence – a person or document for example. A reviewer of such estimates might be a skeptic, and want to check the evidence first hand. We can even rate the *quality of the basis for the estimate* using the 'credibility index' (say 0 for no credibility at all, and 1.0 for 100% credibility). Notice we can use the credibility-rating number to modify, by multiplication, the initial estimate, in the direction of a *more pessimistic* estimate. Better to be safe.

Principle 8: Designs must be evaluated with respect to uncertainty, and the level of risk you want to take.

I also like to get a simple estimate of the *cost* of the design, at least to become conscious of cost extremes.

Table 2. A simplified example of using an impact estimation table to collect data about a single performance attributes. In this case, the performance attribute is 'Learning', which has a target level of 10 minutes. There are four design idea candidates (For example, 'On-line Support' is the tag of one design idea). We need to repeat this process for all other critical performance requirements. This is difficult because of lack of facts about most designs, in most dimensions. But the difficulty usefully makes us formally aware of design risks, and consequent project risks – which we can decide to mitigate by investigation, contracting, design or re-design.

	On-line Support	On-line Help	Picture Handbook	On-line Help + Access Index
Learning Past: 60 minutes <-> Goal: 10 minutes				
Scale Impact	5 min.	10 min.	30 min.	8 min.
Scale Uncertainty	±3 min.	±5 min.	±10 min.	±5 min.
Percentage Impact	110%	100%	60%	104%
Percentage Uncertainty	±6% (3 of 50 minutes)	±10%	±20%	±10%
Evidence	Project Ajax achieved 7 minutes	Other Systems	Guess	Other Systems + Guess
Source	Ajax Report, Page 6	World Report Page 17	John B.	World Report Page 17 + John B.
Credibility	0.7	0.8	0.2	0.6
Development Cost	120K	25K	10K	26K
Performance to Cost Ratio	110/120 = 0.92	100/25 = 4.0	60/10 = 6.0	104/26 = 4.0
Credibility-adjusted Performance to Cost Ratio (to 1 decimal place)	0.92*0.7 = 0.6	4.0*0.8 = 3.2	6.0*0.2 = 1.2	4.0*0.6 = 2.4
Note: Time Period is two years.	Longer timescale to develop			

6 Resource-Based Optimization

Of course, you do not understand a design idea, if you do not understand its costs. I mean the *entire range* of cost types (for example, effort, time, and money). I mean for the entire system lifespan.

Why do projects consistently run over time and budget, and you never seem to have enough people to do the job? [7]. One reason is that people fail to evaluate the costs of their designs. We do not practice 'design to cost'.

At least, if you have two or more promising design idea alternatives, you should consider using the one with the least impact on your resource budgets.

Principle 9: Design ideas must also be evaluated with respect to the design costs' relation to our finite resources. Don't design what you can't afford.

But, I don't see people doing this in practice. I just see them running out of resources and instead of understanding that it might come from poor design practices, they blame other causes (such as too few resources).

7 Risk-Based Elimination

So, at this point, if you have followed the advice above, you might feel you have picked a winner set of design ideas with high performance impacts at low costs. But this is probably all based on *estimates*. Maybe those estimates are based on thin ice, such as rumor? Maybe experience data says the spread of *possible* actual design impacts on requirement levels is quite wide (like 10 minutes ± 9.9 minutes)? Maybe the 'technology behind the design' is not *that* new, but it has never been tried in *your* 'space vehicle', only in 'bicycles'? Enter the idea of 'risk evaluation'. What is the risk that your design idea, however hot it looks on paper, will not *really* work, or worse will ruin your entire project?

Twelve Tough Questions

1. NUMBERS: Why isn't the improvement quantified?
2. RISK: What's the risk or uncertainty and why?
3. DOUBT: Are you sure? If not, why not?
4. SOURCE: Where did you get that information from? How can I check it out?
5. IMPACT: How does your idea affect my goals?
6. ALL CRITICAL FACTORS: Did we forget anything critical?
7. EVIDENCE: How do you know it works that way?
8. ENOUGH: Have we got a complete solution?
9. PROFITABILITY FIRST: Are we going to do the profitable things first?
10. COMMITMENT: Who's responsible?
11. PROOF: How can we be sure the plan is working?
12. NO CURE: Is it no cure, no pay?

Fig. 3. Twelve Tough Questions to help strengthen plans. A more detailed treatment of these questions is in a paper at http://www.gilb.com.

So, we need to ask the risk questions about each design idea. My favorite set of risk questions is my ‘Twelve Tough Questions’, given in Figure 3.

List of Principles

1. A design can only be evaluated with respect to specific clear requirements.
2. All designs have performance and cost attributes, but not necessarily the ones you require.
3. The real value of a design to a stakeholder depends partly on the technical characteristics of the design, and partly on the planned, perceived and actual use of those characteristics in practice, over time.
4. It doesn’t matter how good or how cheap a design is, if constraints forbid it.
5. Designs should not be rejected permanently. The reasons for rejection should be clearly documented, the design specification kept; and the rejection possibly re-evaluated later.
6. The major capability of a design is its ability to contribute to required residual performance levels.
7. A design will be best understood in terms of its multiple quantified impacts on your residual requirements.
8. Designs must be evaluated with respect to uncertainty, and the level of risk you want to take.
9. Design ideas must also be evaluated with respect to the design costs’ relation to our finite resources. Don’t design what you can’t afford.
10. The evaluation of a design idea is a continuous process over a series of estimation and validation events. A lot of questions need asking, by a lot of people, and we need many good answers to evaluate a design.
11. The best practical evaluation of design risks is by practical small step integration of the design, with measurement, feedback and analysis of its real performance and costs. Evolutionary evaluation helps us make better decisions about designs than any review committee will ever be able to make.

Fig. 4. A list of the principles presented in this paper

Curiosità: Insatiably curious, unrelenting quest for continuous learning
Dimostrazione:
Commitment to test knowledge through experience, willingness to learn from mistakes.
Learning for ones self, through practical experience
Sensazione: Continual refinement of senses. As means to enliven experience.
Sfumato: Willingness to embrace ambiguity, paradox, uncertainty
Arte/Scienza: Balance science/art, logic & imagination, whole brain thinking
Corporalità: Cultivation of grace, ambidexterity, fitness, poise
Connessione: Recognition & appreciation for interconnectedness of all things and phenomena.
Systems thinking

Fig. 5. Da Vinci’s Principles from How to Think Like Leonardo da Vinci by Michael Gelb. They describe the evolutionary principles for handling risk.

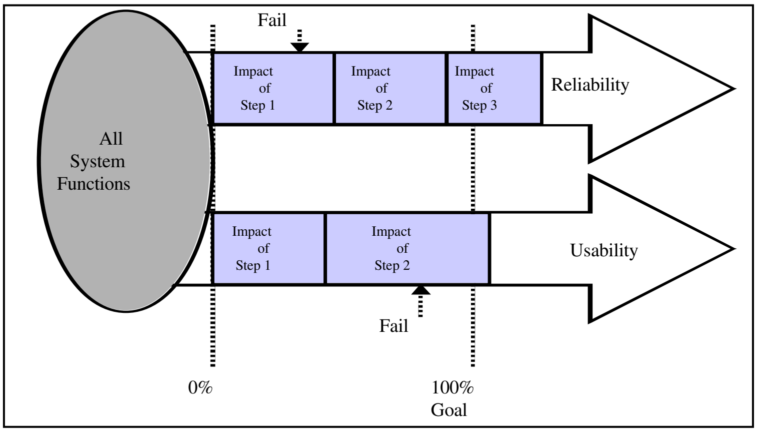


Fig. 6. The step-by-step evolution of designs delivering impact to performance requirements

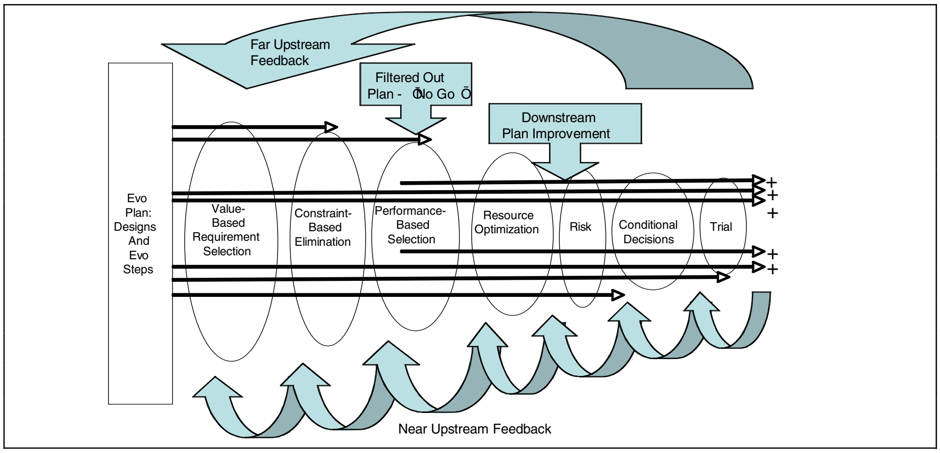

Far Upstream Feedback
Filtered Out Plan - ÒNo Go Õ
Downstream Plan Improvement
Evo Plan: Designs And Evo Steps
Value-Based Requirement Selection
Constraint-Based Elimination
Performance-Based Selection
Resource Optimization
Risk
Conditional Decisions
Trial
+
+
+
+
+
Near Upstream Feedback

Fig. 7. Relevance Control filters start after the QC filters of Rules and Exit make sure we have good presentation. The Relevance Control filters deal with questions of substance: how good is the plan in practice? The QC filters deal with the question, how well is the plan presented? The downstream plan improvements can come from any source, any reason, at any time, or any stage downstream (See also [1] for details of SQC).

We have been asking some of these twelve analytical questions earlier in the design evaluation process above. But some are new. Who is responsible for making it work? Who is responsible if it does not work? Is their money where their mouth is?

I believe, in sharp contrast with the papers and textbooks that I have seen on risk management, that the risk analysis process is something that needs to be *intimately pervasive* in *every single* specification, in every detail of it. It must be part of what all systems engineers do every minute of their working life. Live it and breathe it. Every systems engineering specification has an element of risk – or it would not be termed 'engineering' [8].

We *need*, not to minimize risk, nor to reduce it to zero, but *to be constantly aware of risks.* We need to be constantly looking, waiting to pounce on risk if it shows signs of giving us trouble [9].

Principle 10: The evaluation of a design idea is a continuous process over a series of estimation and validation events. A lot of questions need asking, by a lot of people, and we need many good answers to evaluate a design.

Our systems engineering work should be totally robust so that no matter what happens we have a backup. We have a reasonable way out. We need to be so sensitive to the impacts of our designs that we know when we are threatened. We know early, because we worry early. We try things out early. We keep on measuring early as we make changes and add new designs cumulatively into the system.

We need above all not to trust a probability model of risk analysis. We need to take da Vinci's advice and try things out. See Figure 4. Much of his advice can be seen in the Evolutionary project management model, with its 2% increments, required for measurement, use of feedback, analysis of the feedback, and concept of changing the plan as necessary. We need to use evolutionary step planning to consciously sequence the riskiest elements for early integration and field trialing. Then if there is something wrong, we have lots of time to fix it.

Evolutionary project management (Evo) [1], [4], [5], [10] is one of the greatest devices for risk management and for design evaluation with respect to risk, but Evo never, as far as I can see, made it into a paper or book on risk management, other than my own [9]! Evo allows you to evaluate one design at a time, and to evaluate them cumulatively, one at a time [6].

In fact too many project management people have no clue what Evo really is. However, the US Department of Defense (DoD) finally understood it and adopted it (in 1995 with Mil Std 498 and on), calling it 'Evolutionary Acquisition'.

Principle 11: The best practical evaluation of design risks is by practical small step integration of the design, with measurement, feedback and analysis of its real performance and costs. Evolutionary evaluation helps us make better decisions about designs than any review committee will ever be able to make.

8 Summary

Design evaluation needs a series of processes to determine the best-known design for a specific project. The foundation is a complete, clear and quantified set of requirements, against which to judge the design ideas. The second is a detailed design specification including justifications, assumptions, sources, and expected impacts. The third is the ability to see the expected effects of a set of design ideas, and their total impact on requirements. This initially can be achieved using an Impact Estimation (IE) table. However ultimately a design needs to be proven in practice by evolutionary implementation of the design ideas, while measuring their real cumulative impacts.

References

1. Gilb, Tom: Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, Elsevier Butterworth-Heinemann (Due June/July 2005) ISBN 0750665076. See `http://books.elsevier.com/companions`
2. Gilb, Tom: Agile Specification Quality Control: Shifting emphasis from cleanup to sampling defects. Proceedings of INCOSE Conference, Rochester NY USA (July 2005) Earlier version published as Agile Specification Quality Control. Cutter IT Journal, Vol. 18. No. 1 (January 2005) 35-39. See `http://www.cutter.com` [Last Accessed: April 2005].
3. Gilb, Tom: Real Requirements: How to find out what the requirements really are. Proceedings of INCOSE Conference, Rochester NY USA (July 2005)
4. Larman, Craig: Agile and Iterative Development: A Manager's Guide, Addison Wesley (2003). See Chapter 10 on Evo.
5. Gilb, Tom: Fundamental Principles of Evolutionary Project Management. Proceedings of INCOSE Conference, Rochester NY USA (July 2005)
6. Johansen, Trond and Gilb, Tom: From Waterfall to Evolutionary Development (Evo) or How We Rapidly Created Faster, More User-Friendly, and More Productive Software Products for a Competitive Multi-national Market. July 2005. Proceedings of INCOSE Conference, Rochester NY USA (July 2005)
7. Gilb, Tom: Project Failure Prevention: 10 Principles of Project Control. Proceedings of INCOSE Conference, Rochester NY USA (July 2005)
8. Koen, Billy Vaughn: Discussion of The Method: Conducting the engineer's approach to problem solving. Oxford University Press (January 2003) ISBN 0-195-15599-8. See also `http://www.me.utexas.edu/~koen/` [Last Accessed: April 2005].
9. Gilb, Tom: Managing Your Project Risks in Requirements, Design and Development: Using the Planning Language. Proceedings of INCOSE Conference, Washington DC (2003)
10. Larman, Craig and Basili, Victor R.: Iterative and Incremental Development: A Brief History. IEEE Computer Society (June 2003) 2-11
11. Gilb, Tom: Rule-Based Design Reviews. Software Quality Professional, Vol. 7, No. 1 (2004) 4-13. See website for American Society for Quality: `http://www.asq.org` - member access only for recent papers.
12. Gilb, Tom and Maier, Mark: Managing Priorities: A Key to Systematic Decision Making. Proceedings of INCOSE Conference, Rochester NY USA (July 2005)

The Application of an Object-Oriented Method in Information System Security Evaluation

Qiang Yan and Hua-ying Shu

School of Economics and Management,
Beijing University of Posts and Communications,
Beijing 100876, China
yanqiang@infosec.pku.edu.cn

Abstract. It's essential for critical systems to measure their security status. However, the research on the information system security evaluation still faces many difficulties which are caused by the complexity of the system and the inexplicit relation between the component security and the system security. In this paper, an object-oriented information system security evaluation method is introduced, the security context object model and security evaluation object model are established. These models resolve the current problems and a set of information system security evaluation tools are developed according to these works. The application of the tools is introduced and the deficiencies which need further improvement are also pointed out.

1 Introduction

Information system security evaluation provides the basis of confidence to users. The evaluation results help users to make sure whether the information system is secure enough or the potential risk in operation is acceptable.

China published "Computer Information System Security Protection Classifying Criteria" (GB 17859) [1] in 1999 and adopted Common Criteria (CC) as GB/T 18336. These criteria, including CC, define standards to be used as the basis for evaluation of security properties of IT products and systems. However, the research on information system security evaluation method still faces many difficulties, which include:

- Complexity of information system per se. Information system is the integration of computer or communication hardware, software and firmware for information processing. It includes not only local computing environment and enclave boundary, but also remote terminals and network infrastructure. The topology and application vary from one system to another. The uncertainty of system boundary, complexity of topology and the variety of applications lead to the complexity of security evaluation.
- The relationship between the security of components and the whole system. Information system often comprises components that are developed and evaluated independently. During the evaluation of the system, the relationship between the security of components and the whole system must be clarified firstly. However, current evaluation criteria didn't explain the problem of combination [2, 3].

- The lack of testing methods and tools for security functions. Penetration testing is the mostly used method to evaluate the security of information systems at present. However, unlike security functional testing, which demonstrates correct behavior of the system's advertised security controls, penetration testing is a form of stress testing, which exposes weaknesses in the TCB. Penetration testing is the complement of security functional testing [4] and can't by itself reflect the overall security posture of the information system. Moreover, there still exist controversies about the form of penetration testing results [2, 5].

This paper introduces an object-oriented information system security evaluation method that takes advantage of class, encapsulation and inheritance mechanism of object-oriented technology to solve the above problems.

The remainder of this paper is organized as follow. Section 2 introduces object model of information system security context. Section 3 introduces object model of information system security evaluation. Section 4 describes the realization and application of our method. Discussion about our method and related work are presented in Section 5. Finally our conclusions are outlined in Section 6.

2 Object Model of Information System Security Context

2.1 Security Concepts and Relationships Defined in CC

According to the description of CC [6], information system and its components are assets of their owner. Vulnerabilities may exist in these assets and may be exploited or abused by threats, which cause risk to assets. Countermeasures are imposed to reduce vulnerabilities while they may possess vulnerabilities. Residual risk may remain after the imposition of countermeasures. Periodic risk evaluation is necessary to keep the risk at an acceptable level. Figure 1 illustrates the security concepts and relationships defined in CC.

2.2 Object Model of Information System Security Context

This paper establishes object model of information system security context based on the security concepts and relationships defined in CC. In this model, as illustrated by figure 2, class *Asset*, *Vulnerability*, *Countermeasure* and *Threat* are corresponding to the concepts in CC. Class *Risk_State* extends the concept of risk based on the definition of information system security levels in GB 17859.

2.2.1 Class, Attribute and Method

As illustrated in figure 3, two classes inherit from *Asset*: *Component* and *Application_Service*. Component includes software and hardware such as OS, DBMS, firewall and etc. *Application_Service* denotes the services that system provides to users such as email service, web service and etc. Since application services need the cooperation of multiple components, their security depends not only on the security of components but also on the security of the combination of components. So we create class *Application_Service* to distinguish application services from components.

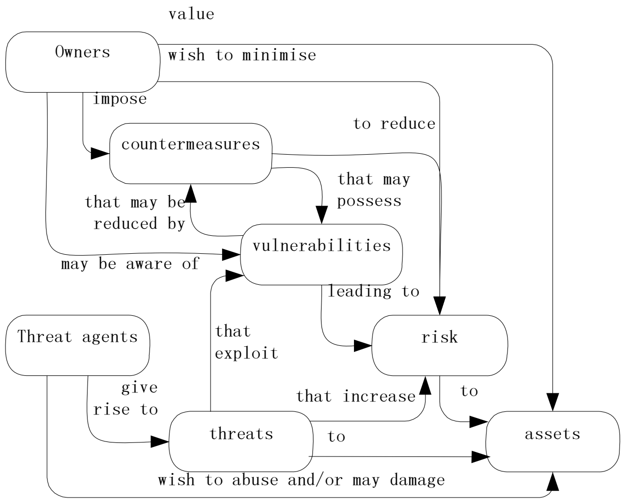


Fig. 1. Security concepts and relationships defined in CC

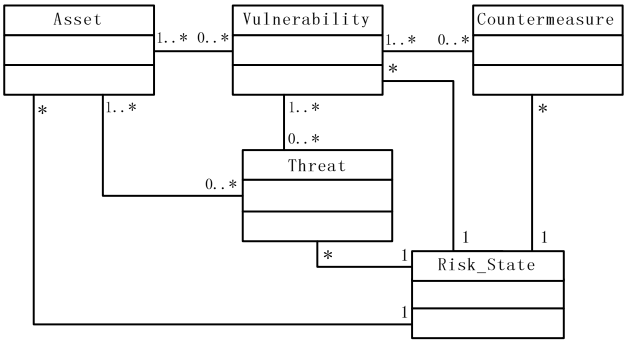


Fig. 2. Object model of information system security context

GB 17859 defines ten security elements and five security levels. These ten security elements are integrity, authentication, discretionary access control, object reuse, audit, label, mandatory access control, trusted recovery, trusted path and covert channel analysis. Attributes of class *Asset* denote the security levels of security elements of asset according to GB 17859. *Component* and *Application_Service* inherit the attributes of *Asset*. These attributes of *Component* are set by the evaluation results of constituent software and hardware, while attributes of *Application_Service* are set during the process of information system evaluation. Besides the inherited attributes, class *Component* adds a new attribute *location*, which points out whether a component belongs to system boundary, computing environment or network. Class *Application_Service* adds two attributes: *components* and *type*. *Components* denote

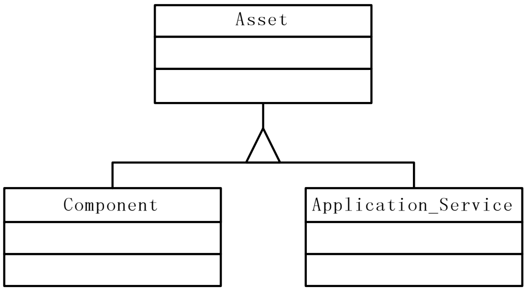


Fig. 3. Inheritance relationship between Asset and Component, Application_Service

those involved in the application service and *type* denotes the type of application such as email service, web service and etc.

Class *Risk_State* extends the concept of risk based on the definition of information system security levels in GB 17859. Traditional risk, including the concept in security context of CC, means the economic losses caused by the damage to assets. It is determined both by the possibility of the occurrence of threats and the consequences caused by the threats. But in the IT field, it is hard to estimate the possibility of the occurrence of threats and the losses caused by the threats can't always be figured by money. For instance, the exposure of cipher may endanger the security of the nation and the occurrence of DoS attack may degrade the reputation of an organization. In GB 17859, different security levels represent the different security states of the security elements as well as of the information system. In our object model of information system security context, security levels reflect the risk state of the information system. The higher the level, the lower the risk. But even the system of highest level doesn't eliminate all security risks. It is just of higher protection capability and lower risk compared with the systems of lower levels.

Class *Risk_State*, *Vulnerability* and *Countermeasure* all have ten attributes corresponding to the ten security elements. In *Risk_State*, these attributes denote the final security levels of security elements of the information system. In *Vulnerability*, these attributes denote vulnerabilities' effects on security elements. While in *Countermeasure*, these attributes mean to which security element countermeasures can provide protection. Moreover, attributes of *Countermeasure* also include the implementing *cost* to support the cost-benefit analysis during the process of risk evaluation. Attributes of class *Threat* include the *possibility* of occurrence, the *capability*, *motivation* and *effect* on security elements of threat.

Class *Risk_State* provides method *evaluation*(), which judges the overall security level of information system according to the attributes of *Risk_State*. Besides, all the classes in the object model provide methods to read and write their attributes.

Class *Asset* and *Countermeasure* are both assets of the system owner. *Asset* is the initial evaluation target. *Countermeasure* is the security control adopted later with the perceiving of risks. But just as described in CC, risk evaluation is a periodic activity; the *Countermeasure* in the first evaluating process will be treated as *Asset* in next turn.

2.2.2. Instance Connection

The instance connections in the object model reflect the following relationships between entities in the real world:

- Assets may possess none or more vulnerabilities;
- A vulnerability may have effect on one ore more assets;
- A vulnerability may be eliminated by none or more countermeasures;
- A countermeasure can eliminate one ore more vulnerabilities;
- Assets may encounter none or more threats;
- A threat may aim at one or more assets;
- A vulnerability may be exploited by none or more threats;
- A threat may exploit one or more vulnerabilities;
- Risk state represents the overall security posture of the information system. Any changes of the entities in the system will cause the changes of the risk state.

3 Object Model of Information System Security Evaluation

Object model of information system security context provides information about the target of evaluation and its environment. This section will introduce object model of information system security evaluation to describe the evaluation actions utilizing the object-oriented technology.

3.1 Class, Attribute and Method

Current security evaluation criteria demand not only vulnerability testing and penetration testing but also security function testing. So we establish six classes: *Correlation_Testing*, *Dependency_Testing*, *Scan*, *Attack*, *Questionnaire* and *Criteria*.

Class *Correlation_Testing* and *Dependency_Testing* analyze the compositional security of components from the aspects of correlation and dependency. Correlation means that components in the information system should keep consistent and cooperative with each other and comply with the unified security policy of the whole system when they enforce their functions. Dependency means that the security of a component, which is deficient in security per se, can be enhanced by other components. During the process of information system security evaluation, the function testing mainly focuses on correlation and dependency between components, while the function testing of components themselves is carried out during the process of products evaluation. Correlation and dependency are dynamic relationships between components and take place when system provides application services to users. Focusing on the process of application services, the testing of correlation and dependency is a supplement of penetration testing and distinguishes the security evaluation of information systems from that of products. To discriminate the correlation and dependency between components locating at different parts of the system topology, class *Correlation_Testing* and *Dependency_Testing* include three categories of attributes:

- Computing environment correlation/ dependency rules
- Boundary correlation/dependency rules
- Network correlation/dependency rules

Each of these categories includes ten attributes corresponding to the correlation and dependency rules of the ten security elements. The methods of class *Correlation_Testing* and *Dependency_Testing* include:

– *rule_substitute*()
– *testing_navigate*()
– *assess*()

When testing the application service, correlation and dependency rules are substituted according to the attribute *location* of the components involved. Final verdict is obtained through the testing navigation.

The methods of class *Scan* and *Attack* are *scan*() and *attack*(), which carry out penetration testing on *Component* and *Application_Service* objects. Attribute of class *Scan* and *Attack* is *type*, which indicates the types of scan and attack such as network scan, system scan, cache overflow and so on.

The methods of class *Questionnaire* include *investigate*() and *assess*(), which investigate the internal and external environments to find out the potential threats to information system by well-designed questionnaire and assess the impact of these threats on the security state of the system. The attribute of class *Questionnaire*, *coverage*, identifies the depth and breadth of investigation, the value of which falls in set {1,2,3,4,5} corresponding to the five security levels of GB 17859.

Class *Criteria* represents the standards for correlation testing, dependency testing, scan, attack and investigating. The attributes of class *Criteria* indicate the security requirements and the methods of *Criteria* provide means to read and write these requirements.

3.2 Instance Connection

During the process of evaluation, class *Correlation_Testing*, *Dependency_Testing*, *Scan*, *Attack* and *Questionnaire* read security requirements from class *Criteria*. *Correlation_Testing* and *Dependency_Testing* send messages to *Application_Service* and *Component* to get the types of application services, the location of constituent components in the system topology and the evaluation results of components themselves. According to the corresponding rules, correlation and dependency between components located at system boundary, computing environment and network are tested respectively and the security of the application services is evaluated. The results are sent to class *Risk_State* and thus the later adjusts the security status of the information system.

Class *Scan* and *Attack* carry out penetration testing to find out the vulnerabilities of the system. *Scan* and *Attack* synthesize the information of vulnerabilities with corresponding impacts and security requirements, which are read from class *Vulnerability* and *Criteria* separately, and send the results to *Risk_State*. Class *Risk_State* adjusts the security status of the information system according to these messages.

During the process of evaluation, the possibility, motivation and the source of threats are investigated in the form of questionnaire and the results are sent to *Risk_State*. Class *Risk_State* adjusts the security status of the information system according to these messages.

Class *Risk_State* receives messages from class *Correlation_Testing*, *Dependency_Testing*, *Scan*, *Attack* and *Questionnaire* and makes the final verdict of the security of information system.

Class *Countermeasure* reads the security state of the information system from class *Risk_State*. According to the protection countermeasures can provide and corresponding implementation costs, class *Countermeasure* recommends security controls to system owner during the process of security improvement.

Figure 4 illustrates the object model of information system security evaluation.

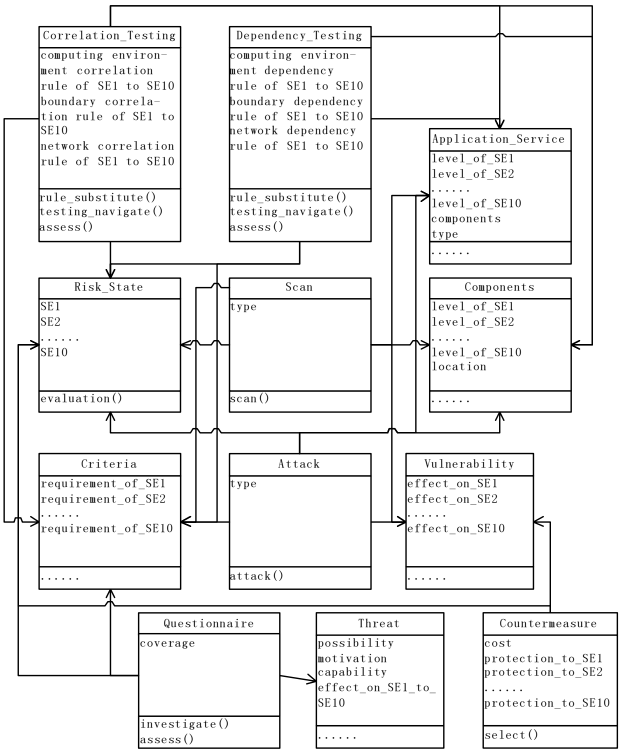


Fig. 4. Object model of information system security evaluation

4 Realization and Application

Based on the object models of information system security context and security evaluation, we developed a toolkit for information system security evaluation. The toolkit now includes eight *Application_Service* objects, i.e. database service, file service, email service, web service, print service, disk service, domain name service and security application service. *Component* objects include most products in market such as firewall, IDS, OS, DBMS, router and etc. *Correlation_Testing* and *Dependency_Testing* objects provide methods, i.e. *rule_substitute*(), *testing_navigate*() and *assess*(), for currently included *Application_Service* and *Component* objects. For example, in a network consisting of a firewall, a database server and other necessary components, the firewall and the DBMS both provide auditing function. The firewall can record the IP addresses of the accessing machines and the DBMS logs the user IDs performing the database operations. When performing *Correlation_Testing*, *testing_navigate*() and *rule_substitute*() will indicate the evaluators to set up rules about how to check if the logs of the firewall and the DBMS can be correlated to depict the complete traces of the user access. Then *assess*() performs the check according to the rules created in the previous step.

Vulnerability objects include information about more than 4,000 vulnerabilities known at present. The severity of these vulnerabilities is often denoted qualitatively such as low, medium and high [7]. However, there are no direct relations between the severity of vulnerabilities and security elements in GB 17859. Vulnerabilities with high severity usually affect more than one security elements, and those with low severity usually affect single one security element while those with medium severity affect from one to more security elements. According to the analysis of vulnerabilities, we associated vulnerabilities with security elements. If any vulnerability has effect on security element n, the attribute of the corresponding *Vulnerability* object, *effect_on_SEn*, is set to one; otherwise the attribute is set to zero. *Scan* and *Attack* objects include the most popular tools available on the INTERNET. With their extensibility, *Scan* and *Attack* objects can verify the survivability of target system under the threat environment of the time. Attributes of *Criteria* object denote the security requirements of GB 17859. The coverage of *Questionnaire* object is corresponding to the requirements of security level one to three of GB 17859. Instances of *Threat* and *Countermeasure* are created during the process of evaluations. We didn't take account of the *cost* of *Countermeasure* objects for simplicity in this toolkit at present. As a representation of system security states, *Risk_State* object reflects the security level of the system. The higher the security level, the lower the security risk.

This toolkit is used in the security evaluations of several companies in Beijing and Hunan province in China. During the testing, we launched the tests from both the internal and the external of the target systems so that we could examine the security of the boundary and the computing environment. Figure 5 illustrates the testing environment.

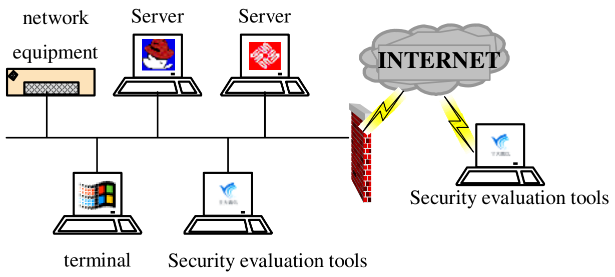


Fig. 5. Testing Environment

The testing shows that:

- The Application Services in the model cover the most conditions during the tests.
- The Attack and the Scan objects meet the testing requirements in the main.
- The correlation and dependency rules need further improvement. Since the network environments vary from case to case, it's hard to figure out all the correlation and dependency rules at present. This work can be improved in the future evaluation experiences.
- The Object-Oriented method is practical in the information system security evaluation.

5 Discussion and Related Work

This paper established object models of information system security context and security evaluation, introduced an object-oriented information system security evaluation method. The main contributions of this method include the following:

- The object model of security context represents the structure and characters of information systems clearly. OO technology simplifies the analysis process and improves the efficiency of security evaluation for complex information systems.
- The object model of security evaluation introduces class *Correlation_Testing* and *Dependency_Testing* to assess the compositional security of components at system boundary, computing environment and network. Focusing on security functions, the testing of correlation and dependency is a supplement of penetration testing and distinguishes the security evaluation of information systems from that of products.
- Class *Scan* and *Attack* encapsulate penetration testing and eliminate the diversity of testing results caused by the diversity of evaluators' experiences. *Vulnerability* associates the results of penetration testing with security elements.
- The object model of security evaluation associates risk management with the evaluation of security levels. It extends the concept of traditional risk and denotes risk state of system by security levels defined in GB 17859.

There are also many other researchers who work on object-oriented security analysis and evaluation methods. Peter Herrmann introduced an object-oriented security analysis and modeling method [8]; J. L. Bramlage proposed an object-oriented risk analysis model [9]; M S Olivier described an object-based version of the path context model [10] to analyze the security of component system. However these methods focus on the analysis of security requirements but not on the evaluation of systems in operation. Bruce Barnett developed a networked object-oriented security examiner, NOOSE [11]. It was used to examine the security of UNIX. Table 1 compares our work with the others.

Although object-oriented analysis and design have been widely adopted in the field of software engineering, the researches on object-oriented security analysis and evaluation are still in their infancy. We hope to establish a practical security evaluation method and corresponding tools and procedures by adopting the object-oriented technology.

Table 1. Comparison between the different models

Model	Application Area	Research Object
Peter Herrmann	CORBA based distributed application	Define the evaluation method
J. L. Bramlage	Risk management of information resources	Risk management method
M S Olivier	Access control of the network resources	Security requirements analysis
Bruce Barnett	UNIX system	Create an object model to support multiple algorithm
Our work	Information system security evaluation	Establish the evaluation procedures and tools

6 Conclusion

This paper introduced an object-oriented information system security evaluation method, established security context object model and security evaluation object model and developed a set of information system security evaluation tools according to these models.

Based on the current research, the future work includes:

- Establish more *Application_Service* and *Component* objects to meet the evaluation requirements of all kinds of information systems.
- Formally describe the correlation and dependency rules.
- Improve the method in practice and develop special object models and tools for typical systems such as E-commerce systems and E-government systems.

References

[1] National Criteria of PRC. Computer Information System Security Protection Classifying Criteria (in Chinese). 1999. Available at http://www.infosec.org.cn/fanv/03_22.htm.

[2] A. K. Ghosh, G. McGraw, An Approach for Certifying Security in Software Components. In Proceedings of 21st NIST-NCSC National Information Systems Security Conference, 1998, pp. 42-48.

[3] Jun Han, Yuliang Zheng. Security Characterisation and Integrity Assurance for Software Components and Component-Based Systems. In Proceedings of 1998 Australasian Workshop on Software Architectures, Melbourne, 1998, pp. 83-89.

[4] Clark Weissman. Penetration Testing. Technical report, Naval Research Laboratory, January 1995. NRL Technical Memorandum 5540:082A.

[5] B. S. Yee. Security Metrology and Monty Hall Problem. Available at: http://www.cs.ucsd.edu/~bsy/pub/metrology.pdf, April 2001.

[6] Common Criteria Project Sponsoring Organisations, Common Criteria for Information Security Evaluation Part 1:Introduction and general model, Version 2.1, August 1999.

[7] URL: http://icat.nist.gov/icat.cfm

[8] Peter Herrmann, Heiko Krumm. Object-oriented Security Analysis and Modeling. In Proceedings of 9th International Conference on Telecommunication Systems – Modelling and Analysis, ATSMA, IFIP, Dallas, TX, USA, March 2001, pp. 21-32.

[9] J. L. Bramlage. A New Paradigm For Performing Risk Assessment. In Proceedings of 20th National Information Systems Security Conference, Baltimore, Maryland. Oct. 1997, pp. 565-576.

[10] MS Olivier, SH von Solms. An Object-based Version of the Path Context Model. International Journal of Computer Mathematics, 49, 3&4, 1993, pp. 133-144.

[11] Bruce Barnett. NOOSE – Networked Object-Oriented Security Examiner. In Proceedings of the 14th Systems Administration Conference (LISA 2000), New Orleans, Louisiana, USA, December 3-8, 2000, pp. 369-378.

Towards a Cyber Security Reporting System – A Quality Improvement Process

Jose J. Gonzalez

Faculty of Engineering and Science,
Research Cell "Security and Quality in Organizations",
Agder University College, Groosveien 36, NO-4876 GRIMSTAD, Norway
Phone: +47 37 25 32 40, Fax: +47 37 25 30 01
Jose.J.Gonzalez@hia.no
http://ikt.hia.no/sqo

Abstract. IT-security lacks the equivalent of an Air Safety Reporting System. Yet, the current trend to outsource security processes might be the birth of a Cyber Security Reporting System – CSRS. A necessary condition for providers of security services to evolve toward a CSRS is successful quality management. The increasing demand for "fire-fighting" – deriving from the growth in number and sophistication of attacks and the decline in the expertise of the average system administrator – pushes farther and farther away from "fire-prevention." But growth of insight, and its codification and communication are prerequisites for even the most rudimentary CSRS. Studies show that few attempts to implement quality improvement processes succeed; yet, successful quality management provides decisive competitive advantage. System dynamics studies of quality management have identified causes of implementation failure and provided guidance for success. Transferring these lessons to security service organizations is a promising path toward the vision of a CSRS.

1 Introduction

Evidence for the benefits of safety and security reporting systems is found in many references. In his seminal book about organizational accidents, Reason states that «many highly effective reporting programmes [do] exist». Reason describes in detail two instances of an 'Air Safety Reporting System'[1] followed by a discussion of successful reporting programs in other domains that have used aviation reporting systems as point of departure and template (cf. [2] p. 196ff). Unfortunately, cyber security (including cyber security in IT-dependent critical infrastructures) are not examples of successful reporting programs. One could quote many papers and books lamenting that the scarcity and incompleteness of (most) security incident data are

[1] The name Air Safety Reporting System is generic; instances of it are e.g. NASA's Aviation Safety Reporting System (ASRS) or the British Airways Safety Information System (BASIS). For one of the first discussions of the benefits Air Safety Reporting Systems see ref. [1], p. 168-169.

hampering progress. A particular and passionate statement (ref. [3], p. 391ff.) compares the frustrating situation for cyber data reporting with the success of 'Air Safety Reporting Systems'.

There should be little doubt about the need to improve reporting of cyber security data – intrusion attempts, successful intrusions, incidents of all kinds, DDoS, etc –, followed by analysis and sharing of insights. True, the numerous computer emergency teams (CERTs) and computer security incident response teams (CSIRTs) around the world have established cyber security reporting systems of sorts. But nearly two decades after their emergence, distinguished experts from the nestor organization among them, the CERT® Coordination Center, acknowledge that systematically collected data on cyber attacks is not generally available (cf. §1.3 Cyber Data Restrictions of ref. [4]). [2]

Reason (op. cit. p. 197) acknowledges that the implementation of critical incident and near-miss reports is not easy. Such task is indeed of daunting difficulty for cyber security. This lack of availability stems from three basic causes: Attackers generally act to conceal their attacks; defenders gather data on attacks for narrow purposes; organizations controlling information assets rarely share data on attacks. First, cyber attacks are in general the more successful, the more unexpected they are. Hence, attackers conceal as much information as possible in order to preserve the utility of their tools; as a result defenders only capture incomplete information on the methods and objectives of the attackers. (An exception are honeypots and honeynets, cf. [5, 6].) Second, defenders of information assets rarely have the capacity and the know-how to collect detailed attack information; in addition, they often are overburdened. Data are only collected if needed for a specific purpose, such as forensic needs or for legal proceedings. Also, collected data is normally stored in ad-hoc formats for the intended purpose – it is rarely stored in generally accessible databases. Third, attack data is rarely shared, and if so, often only in vague terms. Sharing of information may be precluded by the rules of evidence in a criminal prosecution. Often, data is withheld for fear of bad publicity, etc. When detailed data are shared, in most cases restricted use agreements hamper their availability to the research community.

Unfortunately, recent trends seem to indicate a worsening of the situation: Intruders have progressively improved their tools and it has become more difficult to detect an attack. This has a *direct* negative impact on the availability of cyber data. But there is an *indirect* negative effect on availability (both of quantity and quality) stemming from the huge increase in quantity and sophistication of the attacks. Defenders are increasingly overburdened, damage amount is increasing and neither data collection nor smart prevention and detection are becoming easier.

Accordingly, the vision of a Cyber Security Reporting System – CSRS for short – of a scope comparable to an 'Air Safety Reporting System' would appear very distant, nearly utopian. This paper argues that a CSRS might be a realistic vision nevertheless, provided more attention is given to quality improvement processes in entities dealing with incident prevention and handling. It is argued that there are some key elements that – if combined – could lead to more comprehensive and effective cyber data collection and analysis, gradually approaching a CSRS of scope comparable to an Air Safety

[2] The description of cyber data restrictions in ref. [4] was provided by CERT/CC co-authors Dawn Cappelli, Andy Moore and Tim Shimeall.

Reporting System. The most important is that cyber security agents, whether individuals (CISO, CIO) or teams (e.g. CSIRTs), find the right balance between responding to incidents and learning from incidents – this requires a successful quality improvement process to manage the process, as well as improved data collection and analysis to permit learning from incidents. With quality improvement and learning from incidents in place, the trend to outsource security processes will create larger arenas for sharing data and insight, including the development of new methods and tools. For specificity, this paper uses (external) CSIRTs as focal point. First, it is argued that external CSIRTs are good candidates for discussing outsourced security processes from this paper's perspective. Second, evidence is provided that attack and defend trends inhibit the performance of CSIRTs. Third, a simple conceptual model suggests that the performance stress on CSIRTs might force CSIRTs to work hard to the detriment of working smart ("learning from incidents"). The model also shows that CSIRTs can be locked into a "capability trap" – that is a stable underperforming mode. Decisions leading to an underperforming CSIRT are likely to yield a deceitful, transient improvement to begin with (a "better-before-worse" situation). Conversely, the model indicates that the desired mode of a well-performing CSIRT is likely to require a "worse-before-better" stage. Even when managers persevere through the initial "worse" stage to achieve superior performance, shortcuts during stress situations might destabilize CSIRTs toward an underperforming mode.

The perspective of a CSRS is in the sustained operation of working smart and in the diffusion of insights to ever growing circles.

1.1 Outsourcing Security Processes

Cyber security is a very complex field and experts are in great demand. With more and more aspects of our life occurring in cyberspace the demand for security expertise is rapidly growing. Very few organizations have sufficient resources or motivation to have a full cyber defence capacity in-house. Thus, organizations find it necessary to outsource security processes. The constantly changing threat – with the number and sophistication of email, virus, and network-based attacks growing each year – will probably reinforce the current trend to outsource security processes.

Outsourcing of security processes triggers the appearance of informal arenas for sharing of cyber security information. After some transitory phase the arenas develop to fora or institutions. Their very existence and ubiquity promotes sharing of information – within limits – but the constraints will become less restrictive as progress on protocols and data processing activities increases the utility of data for improving security, while protecting the legitimate interest of owners of information sources. Expertise will be shared too – in workshops and conferences (often dedicated to specific security processes, such as the annual FIRST[3] conferences). Generic tools for categories of security processes will be developed – the advantage that the tools provide will facilitate standardization and availability of security data on which they operate. The increasing benefits of the outsourcing security processes (and the accompanying sharing of data) are likely to promote trust as well as further outsourcing and sharing of cyber security data. It is a slow process, but it is happening already.

[3] FIRST is the global Forum for Incident Response and Security Teams.

CSIRTs (Computer Security Incident Response Teams) are of particular interest, since they are probably the most ubiquitous category of outsourced security processes with an arena for diffusion of knowledge to a larger community. The goal of a CSIRT is to control and minimize any damage, preserve evidence, provide quick and efficient recovery, prevent similar future events, and gain insight into cyber threats against a "constituency" of independent organizations.[4] A CSIRT is by definition a clearinghouse for security incident data within the scope defined by its constituency. The TERENA Technical Programme has established the TF-CSIRT Task Force with goals that include providing a forum for exchanging experiences and knowledge and promoting common standards and procedures for responding to security incidents.[5] With such auspices CSIRTs might be seen in retrospect as a significant step towards a future Cyber Security Reporting System – CSRS. Because of this, and for specificity, the remainder of this paper is dedicated to CSIRTs.[6]

Note that a recent CERT/CC study on the state of practice of CSIRTs [7] supports our expectations, at least for a particular domain of cyber security (incident handling): 1) A large increase in the number of incident response teams over the past four to five years before the study (i.e. since 1998) – i.e. a growth in outsourced security services (op. cit. p. 131); 2) the goal to establish standards, and to develop and utilize a common and easy-to-use mechanism for sharing of data between teams and the synthesis of collected data (op. cit. p. 133); 3) the intention to develop generic tools for use in incident handling (op. cit. p. 135.).

2 Attack and Defence Trends

Trends since the early 1990s indicate that the sophistication of attack tools is increasing while the required individual know-how to deploy those tools is decreasing (Fig. 1).[7]

In the 1980s intruders were system experts with a high level of expertise and they personally constructed the methods for breaking into systems. Today, anyone can attack a network using intrusion tools and exploit scripts from the "public domain" that capture known methods of attack. While experienced intruders are getting smarter, as demonstrated by the increased sophistication in the types of attacks, the knowledge required on the part of novice intruders to copy and launch known methods of attack is decreasing.

In the early era of cyber attacks, intruders manually entering commands on computers could access tens to hundreds of systems; today, intruders use automated tools to attack thousands to tens of thousands of systems – and nothing prohibits the access to hundreds of thousands or even millions of sites. In 1980s, it was relatively

[4] Strictly speaking, this statement concerns external CSIRTs. Broadly speaking there are two kinds of CSIRTs, internal and external. An internal CSIRT is a unit within an organization.

[5] TERENA stands for Trans-European Research and Education Networking Association. For details about the task-force, cf. http://www.terena.nl/tech/task-forces/tf-csirt/.

[6] CSIRT is a generic name; cf. http://www.first.org/about/organization/teams/ for examples of existing CSIRTs.

[7] Graphic © Copyright 2004 Carnegie Mellon University. Reprinted with kind permission of CERT®/CC. CERT®/CC is a registered trademark and service mark of Carnegie Mellon University.

straightforward to determine if an intruder had penetrated the system and understand the damage done. Today, intruders are able to totally hide their presence by, for example, disabling commonly used services and reinstalling their own versions, and erasing their tracks in audit and log files. In the beginning of the cyber attack era, denial-of-service attacks were rare and not considered serious. Today, for an increasing number of organizations that operate electronically, a successful denial-of-service attack can put them out of business. Unfortunately, these types of attacks are becoming more frequent. For more details, cf. [7] §3.8 Changes in Intruder Attacks and Tools, p. 107ff. Note also that the rate at which new vulnerabilities are discovered continues to increase (op. cit. p. 111).

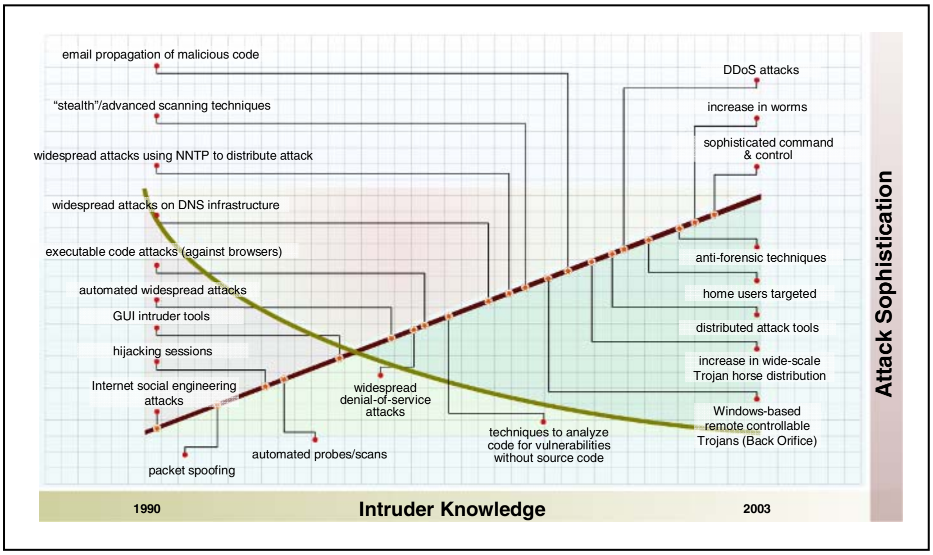


Fig. 1. Attack sophistication vs Intruder Knowledge. © Copyright 2004 Carnegie Mellon University. Reprinted with permission of the CERT® Coordination Center.

Not surprisingly, CSIRTs have problems in coping with the increasing flood of incidents. Killcrece et al. state in the seminal CERT/CC study (ref. [7], p. 77): «Because of the amount of detailed work done by incident handlers and the increasing work loads, many of the authors of the books and articles reviewed in the literature identified staff burnout as a problem for CSIRTs.» Further: «As the volume of incident and vulnerability reports continue to rise, and the automation and speed of many attack tools continue to increase, CSIRT and information security staff members now have less time to react to new threats.» (Op. cit. p. 112).

Summarizing: The increasing number of attacks and their sophistication has increased the workload in CSIRTs and it is becoming overwhelming, implying a wide range of internal problems, such as insufficient funding, inadequate management support, shortage of trained incident handling staff, lack of clearly defined mission and authority, and lack of coordination mechanisms. (Op. cit. p. 126).

In the following section it is argued that this stressful situation is a main obstacle for the reporting of cyber security data, learning from incidents and for sharing of insights.

3 CSIRT Services – Status and Shortcomings

The CERT/CC report on organizational models for CSIRTs classifies services provided by Computer Security Incident Response Teams as reactive services, proactive services and security quality management services. Reactive services are the core component of CSIRT work; they are triggered by incidents events or requests – those services can be compared to fire-fighting activities. Proactive services target preparation, protection and securing constituent systems. Security quality management services «augment existing and well-established services that are independent of incident handling and traditionally performed by other areas of an organization such as the IT, audit, or training departments.» (Ref. [7] p. 65.) While proactive services directly reduce the number of incidents, security management services do indirectly so (cf. [8] p. 14-15.) Proactive services and security quality management processes can be compared to fire-preventing activities.

The CERT/CC report on the state of the practice of CSIRTs [7] gives much evidence that most current CSIRTs still are incipient, that their services are reactive in nature or still striving to achieve maturity: Computer security must be proactive to be successful – being reactive is no longer sufficient (p. 131); incident response is still an immature field and there are few standards (p. 131-132, 133); there are no consistent structure or set of services for a CSIRT (p. 132); There is no commonly used taxonomy for incident response and computer security terminology (p. 132); employees who are trained and experienced in incident response techniques and practices are difficult to find (p. 132); few tools addressing the specific needs of CSIRTs are readily available (p. 132); shortcomings in best practices (p. 132); etc.

One might safely conclude that in many CSIRTs reactive fire-fighting dominates to the detriment of proactive work and security quality management services. To release the potential of CSIRTs in security prevention and their ability to evolve toward Cyber Security Reporting Systems a paradigm shift is needed. In the next section it is argued that the clue is a successful quality improvement program in CSIRTs.

4 The Need for Quality Improvement Processes in CSIRTs

A very interesting parallel to the situation in CSIRTs occurs in quality improvement programs: Despite vast investments – in the USA in the order of hundreds of billions USD per annum – few efforts to implement such programs yield significant results. But enterprises that succeed to implement total quality management programs outperform their competitors [9-12]. A team at MIT's Sloan School of Management led by professors Nelson Repenning and John Sterman has conducted about a dozen brilliant in depth studies of the "quality improvement paradox" in several sectors (telecommunication, semiconductors, recreational products, chemicals, oil, automobiles). System dynamics models based on detailed data are able to capture the essentials of the quality improvement process (see the award-winning paper [13] and references given in endnotes 4-5 therein). If the methods and lessons from the MIT studies can be applied to CSIRTs, a major improvement in their performance might be achieved.

Several projects in our research cell "Security and Quality in Organizations" are concerned with the dynamics of CSIRT management (refs. [14, 15] are results from two of them; in addition there is a recently begun project, AMBASEC,[8] related to incident response and management in the context of eOperation in offshore oil and gas fields). Of particular interest is the PhD project of Johannes Wiik, which is concerned with the management of CNF-CERT, a state-of-the art external CSIRT. Preliminary results with data provided by K.-P. Kossakoswki from CNF-CERT suggest indeed that the MIT approach to quality improvement processes is relevant for CSIRT management as well [14, 16]. The ultimate goal is to derive best practice recommendations in order to improve a generic CSIRT's ability to provide processes and security quality management services. The potential of the approach can be illustrated with an adapted version of a basic, qualitative system dynamics model developed by Repenning and Sterman. The model explains the core behaviour of the dynamics of quality improvement processes; it has four feedback loops depicting the trade-offs between working hard vs. working smart (op. cit., section The Structure of Improvement, p. 66 ff). In the case of CSIRTs, "working hard" corresponds to routine coping with the flood of incidents, while "working smart" is tantamount to learning from incidents – i.e. distilling lessons; proactive work leading to preventive measures; developing tools that improve performance; sharing of insights; promoting better security management; etc.

Figure 2 shows the CSIRT version of Repenning and Sterman's basic model. The four feedback loops determining the performance of the CSIRT are:

- 'B1: COPING WITH INCIDENTS.' This is a balancing loop consisting of the variables '*Performance gap*', '*Pressure to handle incidents*', '*Time spent on incident response*' and '*Actual CSIRT performance*';
- 'B2: LEARNING FROM INCIDENTS.' This is a balancing loop consisting of the variables '*Performance gap*', '*Pressure to improve CSIRT capability*', '*Time spent on improvement*', '*Development of CSIRT capability*', '*CSIRT's capability*', '*Prevented incidents*' and '*Actual CSIRT performance*';
- 'B3: SHORTCUTS.' This is a balancing loop consisting of the variables '*Performance gap*', '*Pressure to handle incidents*', '*Time spent on improvement*', '*Time spent on incident response*' and '*Actual CSIRT performance*';
- 'R1: REINVESTMENT.' This is a reinforcing loop consisting of the variables '*Performance gap*', '*Pressure to improve CSIRT capability*', '*Time spent on improvement*', '*Development of CSIRT capability*', '*CSIRT's capability*', '*Prevented incidents*' and '*Actual CSIRT performance*'.

The plus and minus signs in the diagram refer to the polarity of the causal influence: A plus sign indicates a causal influence in the same direction; a minus sign indicates an influence in the opposite direction.[9]

[8] AMBASEC (A Model-based Approach to Security Culture).

[9] For a more accurate definition of causal link polarity, see ref. [14] p. 139.

A rising number of security incidents (*Attacks*) widens the '*Performance gap*' (the difference between desired and actual CSIRT performance). CSIRT management has to choose the right balance between two options, COPING WITH INCIDENTS and LEARNING FROM INCIDENTS. LEARNING FROM INCIDENTS is in principle the smarter option because preventive measures, better tools, improved security culture, etc., yield enduring change and enhance the power of direct efforts. COPING WITH INCIDENTS is constrained by available staff time and resources, but it has the (dangerous) attraction of yielding immediate results. Unfortunately, the better option LEARNING FROM INCIDENTS, involving development and deployment of smart solutions, takes time to develop and to come into play (the time delay is shown by the delay mark //). There is a trade-off between both options, since '*Time spent on incident response*' and '*Time spent on improvement*' add up to total available time.

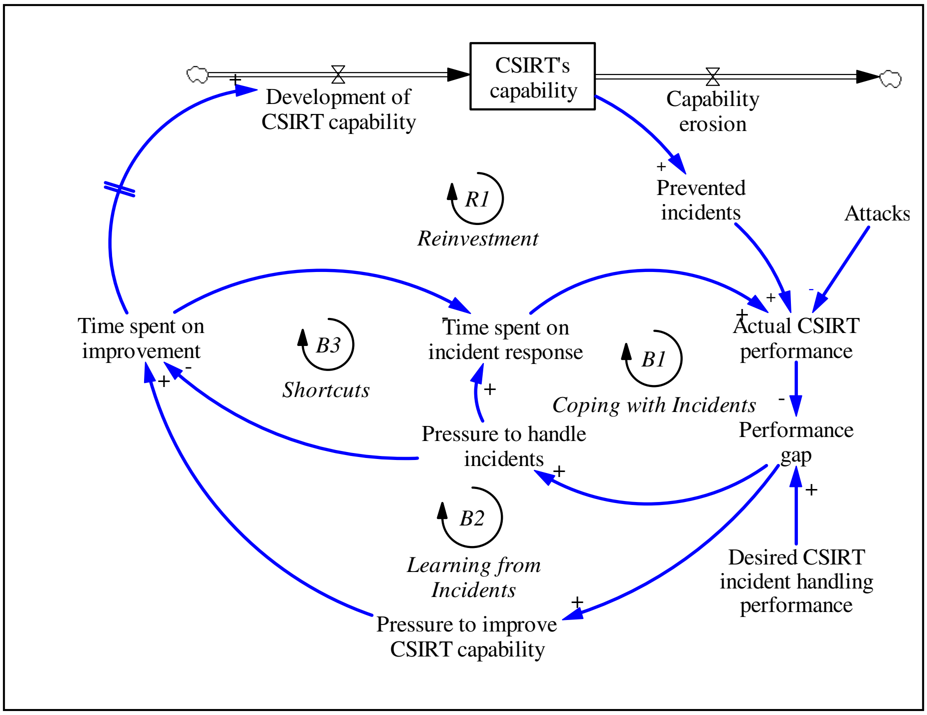

CSIRT's capability
Development of CSIRT capability
Capability erosion
Prevented incidents
Attacks
R1
Reinvestment
Time spent on improvement
B3
Shortcuts
Time spent on incident response
B1
Coping with Incidents
Actual CSIRT performance
Performance gap
Pressure to handle incidents
B2
Learning from Incidents
Desired CSIRT incident handling performance
Pressure to improve CSIRT capability
+
-

Fig. 2. Basic system dynamics model of CSIRT performance

Working under the pressure to respond to an increasing stream of incidents, most managers of CSIRTs would react by increasing '*Pressure to handle incidents*'. The resulting balancing feedback loop 'B1: COPING WITH INCIDENTS' counteracts the widening performance gap and some transient improvement is seen.

CSIRT management could opt to increase '*Pressure to improve CSIRT capability*'. There is a dilemma, because '*Time spent on improvement*' does not yield immediate results; it takes time for piecemeal improvements (depicted as the flow variable '*Development of CSIRT capability*') to add up to sizable '*CSIRT's capability*' that tends to persist. (The persistence of '*CSIRT's capability*' means that it is a stock variable – shown as a rectangle – that cumulates the flow, the improvements over time.) Ultimately, capability does erode over time (as knowledge, routines, tools, etc. become obsolete); this is depicted by the flow variable '*Capability erosion*'. Improved '*CSIRT's capability*' rises '*Prevented incidents*' and so does '*Actual CSIRT performance*'. Summarizing: The balancing feedback loop 'B2: LEARNING FROM INCIDENTS' acts also to close the '*Performance gap*', it gives quite persistent results, but it takes time.

The feedback loop 'R: REINVESTMENT' is reinforcing; the conundrum is that it reinforces whichever strategy management chooses. If management gives too much priority to reactive work, perhaps because the swelling flood of security incidents forces them to, the REINVESTMENT loop increases '*Pressure to handle incidents*' even more at the expense of CSIRT's capability (in other words the REINVESTMENT loop makes the COPING WITH INCIDENTS feedback loop the dominant one, i.e. the CSIRT is likely to lock into reactive fire-fighting at the expense of becoming smarter). If management instead decides to increase '*Pressure to improve CSIRT capability*', the REINVESTMENT loop facilitates further improvements of '*CSIRT's capability*' (i.e. the REINVESTMENT loop makes the LEARNING FROM INCIDENTS loop the dominant one and the CSIRT becomes more effective at handling and preventing incidents).

Figure 3 shows the results of two simulations showing how a hypothetical CSIRT would react to management opting for more "Coping with Incidents" vs. more "Learning from Incidents" (working harder vs. working smarter). In both cases the CSIRT begins in the same stationary state, but a sudden increase in intrusions forces management to either handle more incidents or to invest in more capability development (via "Learning from Incidents"). The first simulation shows the CSIRT's response to emphasis on "Coping with Incidents." As more pressure on handling incidents increases, effort to handle incidents does so too. Time spent on improving CSIRT capability falls quickly, but CSIRT capability stays the same for a while. Hence, the actual CSIRT performance, in terms of how many incidents are handled per time unit, rises without delay. But the gain of working harder to cope with incidents is transitory. In contrast, if management decides to invest in CSIRT capability development through learning from incidents, CSIRT performance falls transiently until the gain from enhanced capability results in sustained superior performance.[10] Choosing working harder (more coping with incidents) gives a "better-before-worse" response; opting for working smarter (more capability development through learning from incidents) yields a "worse-before-better" situation. CSIRT management can be lured by short-lived gains and become locked in an underperforming situation, or it can prematurely give up developing sustainable CSIRT capability. The term "capability trap" (coined by Repenning and Sterman – op. cit.) is a metaphor for this double peril.

But there is more to it. Even when management has opted for more LEARNING FROM INCIDENTS, there is a substantial risk that emergency would force the staff to temporarily reduce '*Time spent on improvement*' and to increase '*Time spent on*

[10] In the conceptual model behind Figure 3 it is also assumed a smooth transition to more effort on capability improvement (r.h.s.), whereas it is assumed that the transition to more incident handling can be implemented very fast (l.h.s.).

incident response'. The resulting 'B3: SHORTCUTS' loop is again a balancing loop that helps meet short time objectives – but the time won by cutting corners reduces the capability of the CSIRT. In other words, the SHORTCUTS loop can in a subtle way lead away from the best strategy and lock CSIRT performance in the "capability trap."

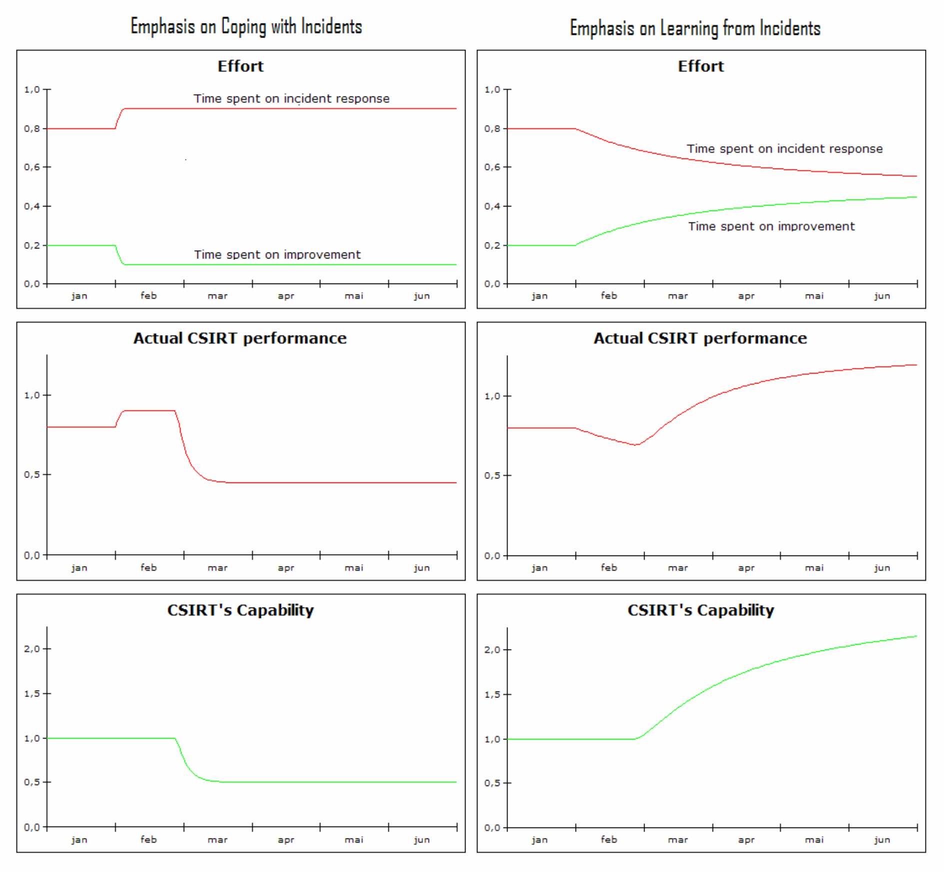


Fig. 3 Simulations of CSIRT response to emphasis on Coping with Incidents vs. Learning from Incidents (Time scale is arbitrary)

5 Concluding Remarks

At first sight, the vision to establish a Cyber Security Reporting System (CSRS) in the spirit of 'Air Safety Reporting Systems' seems too distant to be a guide for current endeavours. But the vision might not be that distant: In this paper it is argued that the outsourcing of security processes – particularly of incident response handling – might be a first step toward a CSRS. A necessary condition for this to happen is that providers of outsourced security services improve their capability to do proactive work and to deliver security quality management services. The challenge is analogous to quality improvement processes. This paper suggests that CSIRTs are a promising first venue for quality improvement.

The analysis derived from the pioneering work on quality improvement processes lead by Repenning and Sterman at MIT's Sloan School of Management makes it promising to transfer the methods and to transfer their approach to quality improvement processes in CSIRTs: If the capability of CSIRTs (of which there are hundreds in the world) improves significantly, there would be a multiplier effect concerning better data mining and use of incident data to improve organizational security and survivability. Thus, directing the performance of CSIRTs toward more proactive work and improved security services would have strong leverage when it comes for progress in data mining and downstream activities from cyber security data (analysis, modelling, insight, dissemination of results, etc).

To find out the right balance between reactive work and learning from incidents is not a trivial task; nor is the implementation of the right strategy – once found – trivial. Among the problems facing management is the dilemma that – in order to achieve sustainable gains in CSIRT capability – the CSIRT's performance is likely to decrease before it improves. Indeed, to improve CSIRT capability one must invest the existing limited resources into learning from incidents, i.e. to redirect efforts from response to quality improvement – an investment that takes time to yield visible results. And vice versa: Performance improves transiently if management enforces more reactive work to cope with the swelling flood of security incidents – in the meantime CSIRT capability erodes and a strong negative outcome in performance ensues. Management can fall into the capability trap in two ways: 1) Emphasis on coping with incidents yields initially deceitful promising results, but subsequently CSIRT performance gets locked in an underperforming mode; 2) emphasis on learning from incidents does establish superior and sustained performance, but inadvertent recourse to more and more shortcuts during stressful periods of incident response destabilizes performance.

The capability trap is hard enough to avoid in enterprises under tough competition pressure; the exponential rise of security incidents and the growing sophistication of attacks puts CSIRTs (and probably most other organizations providing security services) under an even greater pressure; the basic, conceptual system dynamic model discussed above would suggest that avoiding the capability trap in the cyber security case is an even tougher task than in, say, manufacturing companies. A central target in the projects of our research cell Security and Quality in Organizations is to suggest specific CSIRT management policies that prevent the capability trap and, hence, a degrading effectiveness of such organisations. Achieving sustainable superior CSIRT performance is expected to promote better security data mining and learning from such data.

CSIRT is a generic name and instances of CSIRTs are found in critical infrastructure (defence, finance, energy sector, etc.). The PhD project of Johannes Wiik and the AMBASEC project are just a beginning – though a promising one. More studies will be needed, since the topic is complex and CSIRTs occur in many shapes and configurations. For further progress to occur it is crucial that incident response teams engage in quality improvement processes – a modest first step would be to interact with our team (i.e. comment and criticize our approach, to discuss cooperative projects, etc.).

Although this paper has selected external CSIRTs as specific case for the vision to CSRS through quality process improvement, much of the arguments presented should be of generic validity for other kinds of outsourced security processes.

Improved cyber security is closely related to dependability of infrastructures that depend critically on information technology. This perspective is beyond the scope of this paper but it is a central aspect of the collaboration of the AMBASEC project of our research cell Security and Quality in Organizations with SINTEF's IRMA project. The target of the collaboration is to improve information security in the oil and gas industry, for individual enterprises and as part of critical infrastructure, by improving incident response management.

Acknowledgement

The work leading to this paper is part of the AMBASEC project that is funded by the Research Council of Norway under the IKTSoS program.[11] The idea to write this paper surfaced when attending the IKTSoS workshop at Oslo Airport Hotel 1-2 March 2005. We express our gratitude to the Research Council of Norway for arranging and financing the IKTSoS workshop.

The paper has greatly benefited from discussions with PhD fellow Johannes Wiik and post doctoral fellow Dr. Agata Sawicka. We thank also project secretary M.Sc. Maren S. Assev for her critical comments and stylistic corrections.

We also thank the CERT®/CC, Carnegie Mellon University, for permission to reproduce Fig. 1 and to adapt the explanatory text.

References

1. Perrow, C., *Normal accidents: living with high-risk technologies*. 1999, Princeton, N.J.: Princeton University Press. Original pub.: New York: Basic Books, c1984.
2. Reason, J., *Managing the Risks of Organizational Accidents*. 1997, Aldershot, Hants, UK: Ashgate Publishing Ltd. 1997.
3. Schneier, B., *Secrets and Lies: Digital Security in a Networked World*. 2000, New York: John Wiley & Sons, Inc.
4. Andersen, D.F., et al. *Preliminary System Dynamics Maps of the Insider Cyber-threat Problem*. In *Twenty Second International Conference of the System Dynamics Society*. 2004. Oxford, UK.
5. Spitzner, L., *Honeypots: Tracking Hackers*. 2003, Boston: Addison-Wesley Publishing Company.
6. The Honeynet Project, *Know Your Enemy: Learning About Security Threats*. 2 ed. 2004, Boston: Addison-Wesley Publishing Company.
7. Killcrece, G., et al. *State of the practice of Computer Security Incident Response Teams (CSIRTs)*. 2003 [cited 2005 24 February]; Available from: http://www.cert.org/archive/pdf/03tr001.pdf.
8. Killcrece, G., et al. *Organizational Models for Computer Security Incident Response Teams (CSIRTs)*. 2003 [cited 2005 23 February]; Available from: http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03hb001.pdf.

[11] AMBASEC, A Model-based Approach to Security Culture, grant number 164384/V30. IKTSoS is an abbreviation for IKT Sikkerhet og Sårbarhet (ICT Security and Vulnerability).

9. Easton, G.S. and S.L. Jarrell, *The effects of total quality management on corporate performance: An empirical investigation.* Journal of Business, 1998. **71**(2): p. 253-307.
10. Hendricks, K.B. and V.R. Singhal, *Quality awards and the market value of the firm: An empirical investigation.* Management Science, 1996. **42**(3): p. 415-36.
11. Hendricks, K.B. and V.R. Singhal, *Does implementing an effective TQM program actually improve operating performance? Empirical evidence from firms that have won quality awards.* Management Science, 1997. **47**(9): p. 1258-74.
12. Hendricks, K.B. and V.R. Singhal, *Firm characteristics, total quality management, and financial performance.* Journal of Operations Management, 2001. **19**(3): p. 269-285.
13. Repenning, N.R. and J.D. Sterman, *Nobody ever gets credit for fixing problems that never happened.* California Management Review, 2001. **43**(4): p. 64-88.
14. Wiik, J. and J.J. Gonzalez. *Limits to effectiveness of Computer Security Incident Response Teams (CSIRTs).* in *TwentyThird International Conference of the System Dynamics Society*. 2005. Boston, MA: The System Dynamics Society.
15. Sawicka, A., J.J. Gonzalez, and Y. Qian. *Managing a CSIRT.* in *Twenty Third International Conference of the System Dynamics Society*. 2005. Boston, USA.
16. Wiik, J. and K.-P. Kossakowski. *Dynamics of CSIRT Management.* in *Seventeenth Annual FIRST Conference on Computer Security Incident Handling*. 2005. Singapore: FIRST.
17. Sterman, J.D., *Business Dynamics: Systems Thinking and Modeling for a Complex World.* 2000, Boston: Irwin/McGraw-Hill.

Security Research from a Multi-disciplinary and Multi-sectoral Perspective

Atoosa P-J Thunem

Software Engineering laboratory, Institute for Energy Technology,
NO-1751 Halden, Norway
atoosa.p-j.thunem@hrp.no

Abstract. During the era of information technology and within the domain, the topic of security has for many years been perceived of as a "goodness" factor particularly relevant to IT in general and Telecommunications in particular. Nevertheless, rapid application growth of complex Information and Communication Technologies (ICT) in every society and state infrastructure has revealed vulnerabilities that eventually have given rise to serious security breaches. These vulnerabilities together with the course of the breaches from cause to consequence have gradually convinced the field experts that ensuring the security is no longer possible by only relying on the fundaments of the computer science, IT, or Telecommunications. Appropriating knowledge from other domains of science is not only beneficial, but indeed very necessary. At the same time, it is a common observation today that ICT-systems are used everywhere, from the aviation, nuclear, commerce and healthcare domains to camera-equipped web-enabled cellular phones used by your next door teenagers. There ought to be common mechanisms on security analysis and countermeasures against the possible breaches, which are valid for all these domains. This paper advocates the importance of a multi-disciplinary and multi-sectoral security research and analysis, and highlights the European and Norwegian initiatives in that direction.

1 Introduction

During the recent years, technological research within security has evolved from computer and IT security, through cyber and information security and now to the rapidly growing scope of ICT security. During the era of IT and within the domain, the topic of security has for many years been perceived of as a "goodness" factor particularly relevant to IT in general and Telecommunications in particular. In the light of this, the topic of security from a pure technological point of view has been believed to be a function of mainly three variables, the notorious CIA (*Confidentiality*, *Integrity* and *Availability*). In accordance with the increasing complexity of information and communication technologies and their applications and especially within computer security, *Accountability*[1] is also believed to be the fourth

[1] A system's accountability is usually used to address a quality of a system that makes it possible to trace a security breach (related to one or several from CIA) caused by an artefact uniquely to that artefact.

deciding variable [1]. All four variables are mutually related. Nevertheless, the integration of ICT systems into all groups of society infrastructure has seriously challenged the validity of the CIAA belief. Within the ICT community, a common consensus today is that the deciding CIA variables are closely related to factors, which traditionally have not been regarded as of technological nature. The most compelling evidence is the issue of *safety*: While *security in the context of safety* has so far been an issue only within certain industrial domains such as the nuclear field, it has become more relevant today for other areas, e.g., eHealth. Examples of other factors increasing in their importance are trust, (data) protection of personal privacy, user-friendliness [2][3], robustness, maintainability, flexibility, and mobility.

One of the oldest non-technological perceptions of security is from the banking domain. For many years, sociological, financial, political, defence-political, jurisprudential and environmental observations and analyses have been contributing to a non-technological understanding of security. Then again, the observations and analyses made by these areas today cannot deny the major role of insight into technological trends such as the advance of ICT systems on how to understand and deal with security.

The above indicates the inevitable: Ongoing and future security research efforts within various disciplines and application areas cannot be mutually exclusive, if wished to achieve an acceptable level of success. In other words, security research is by nature a multi-disciplinary and multi-sectoral research area.

2 Security as a Dependability Factor and the Related Challenges

Based on the above, it is not far from the (relative) truth to claim that competence in technological pillars of the ICT domain and dependability analysis, gained knowledge and experience within other domains that are relying on and applying ICT systems, and focus on continuously learning from, exploiting and engaging other disciplines and application areas in the efforts within security research all contribute to better understanding of the relationships between the security in one side and other dependability factors in the other side, and hence to more effective and long-lasting countermeasures against possible threats eventually resulting in serious security breaches [4].

To begin with understanding such relationships, the following provides detailed definition of safety and security and their associated risk. The definitions are not only in agreement with the corresponding definitions offered by applied international standards (e.g., IEC 61508), but also are more advanced, as far as the level of detail and clarity of involved terms are concerned.

2.1 Safety, Security and Their Associated Risk

The term ***safety*** is associated with a system's[2] physical condition not being harmed or damaged by its outside environment (including humans). At the same time, a system

[2] A *system* is a compound of interrelated and interconnected entities that function together in order to attain a set of overall goals for the system. Scientifically, a system can be of natural character (e.g., a human being) or human-made (e.g., an oven, a television set, or the nation-wide electricity power network).

contributes to the safety of its outside environment, when the system is able to function and to be used as intended or expected without harming or damaging this environment. Thus, safety is used to express the prevention of unacceptable *risk* of *harm*. Harm and risk are defined as follows [5]:

- Harm is the physical injury, or the physical damage to condition or property of a system or its outside environment, caused by an intended or unintended action or an event.
- Risk is a collective effect (qualitative or quantitative) of the occurrence likelihood of a hazard causing harm and the degree of severity of the harm, given the degree of vulnerability of the system or its environment subject to that harm.

The perception of failure of a safety-related system can vary considerably depending on the application in focus. It is this variation that leads to concepts such as the "level of safety", the "Safety Integrity Level" (SIL), and the “As Low As Reasonably Possible” (ALARP) for a system.

In general, the term safety is more often applied for living beings than, e.g., pure technological systems. Bearing the physical condition and protection of a system in mind, however, the term is equally applicable for all systems.

The term ***security*** is associated with the protection of a system’s *assets* such as the information[3] and information processing resources, from being *threatened* to unintended or intended damage by the system’s outside environment. Thus, a system’s level of security may decline without affecting the system’s level of safety. As an example, the confidentiality of a nuclear scientist’s knowledge carried by its brain may be intentionally disclosed, hence causing the scientist’s security level to decline, without affecting the scientist’s level of safety in any manner.

Of course, a security breach for a system might affect the safety of its outside environment, both in a positive and negative manner. In the context of security, threat and risk are defined as follows [6]:

- Threat is defined as an intended or unintended action or an event that might jeopardise the security of a system.
- A risk is defined as the collective effect of the occurrence likelihood of a particular threat and the degree of severity of the threat (i.e., the potential consequences of the threat, if it did occur), given the degree of vulnerability of the system subject to that threat.

In general, the term security is more often applied for technological systems than living beings. Bearing the assets of a system in mind (i.e., its information and information processing resources), however, the term is equally applicable for all systems that possess information.

Nevertheless, the tradition of relating safety to the living beings and security to technological systems (or “machines”) is still helpful, when addressing the

[3] Thus, the asset can also include *knowledge*, which is a piece of information already declared to have a certain value of use.

relationship between safety and security (such as "security in the context of safety"). The best illustration existing today is perhaps *the three laws of Robotics*[4]:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

2.2 Pertinent Security-Related Issues Subject to Deeper Exploitation

Assuming that the conditions described in the beginning of this chapter are fulfilled, one is able to identify issues that need further exploitation. Four of these are explained bellow.

1. Identification of technological and non-technological factors defining, deciding or relating to the level of security in security-critical systems, society and state infrastructures and processes. A society infrastructure can be the nationwide electricity network, where the relationships among factors such as availability, integrity, maintainability and safety are crucial to identify in order to ensure an acceptable level of security. A state process can be a continuously updated collection of guidelines and means to implement actions to protect the society against the threat caused by international terrorism. Here, it is of paramount importance to clarify factors such as trust, (data) protection of personal privacy, user-friendliness (e.g., of instructions) and robustness (e.g., against vulnerability sources), in order to establish a certain level of belief in security countermeasures.
2. Identification of both overlaps and discrepancies across involving sectors, so that it becomes easier to develop common methodologies and models to deal with security in present and future complex systems, infrastructures and processes, without causing the tailor-made methodologies and models in each sector to become invalid or ineffective.
3. Integration of risk factors into the established security affecting and security related factors addressed above, so that the entire risk management process, including risk analysis, assessment and treatment can be mapped into the development process (lifecycle) of security-critical systems, infrastructures and processes, hence resulting in *risk-informed* development processes with security as their core focus. In practice, this means that there should be a risk model involved as an integrated part of a certain security related factor for, e.g., a modernised ICT system, an updated guideline, or a modified state decision. This factor could be the "accepted" level of data protection of

[4] The three laws of Robotics were established by the father of Robotics, Isaac Asimov whose thoughts and theories on possible patterns of relationships between humans and machines have highly inspired the masters of information and communication engineering as well as human factors engineering.

personal privacy, so that the consequence of its change to other levels in the future can be viewed and studied.

4. Intensified focus on research within communication and traceability of security affecting and security related requirements for all systems and processes used in technological/industrial, sociological, financial, political, defence-political, jurisprudential and environmental applications and sectors, in addition to society and state infrastructures. Joint efforts from different disciplines within this particular area are central in dealing with continuous changes in the requirements for such complex systems, processes and infrastructures, as a response to modernisation and improvement needs, as well as social, economical, environmental, technological and political influences from the world.

3 European and Norwegian Initiatives Towards Security Research

The European Commission's movements and actions in progress indicate a clearly increasing focus on the topic of security. In that respect, a dedicated programme for security research as a part of the 7th Framework Programme is about to form [7]. The prime rationale behind the programme is in fact to better and more efficiently conform to the multi-disciplinary and multi-sectoral requirements for giving a boost to Europe's security research.

In accordance with the above, the Research Council of Norway has launched several initiatives to prepare the research and education community in Norway for better compliance with the requirements set up by the EC.

3.1 The European Security Research Programme (ESRP)

The major difference between the new programme and the EC's security-related activities within the previous and current framework programmes is the programme's particular focus on joint civil-defence research. The aim is to establish an environment for more coherent research towards security for society and infrastructure, so that the risk for terrorism, organised crime, large-scale accidents and natural disasters can be reduced in a more effective manner. The research programme will therefore be multidisciplinary and multi-industrial of nature. The EC undertook in that regard two concrete actions:

- Launching a Preparatory Action in the field of Security Research (PASR), http://europa.eu.int/eur-lex/en/com/cnc/2004/com2004_0072en01.pdf
- Asking a high level Group of Personalities (GoP) to advise on a long-term strategy for European Security Research Programme (ESRP) within the European Union

The PASR has been launched with the first call closed on June 2004 and the second call closing on May 2005. The main objective of the PASR exercise is to bring together the greatest possible number of interested parties, so that a robust community accustomed to working together is established by the time the comprehensive ESRP is launched in 2007.

The GoP's report, http://europa.eu.int/comm/research/security/pdf/gop_en.pdf, was published in March 2004. This report recommends the following:

1. A Community-funded ESRP ensuring the involvement of all Member States should be launched as early as 2007. Its minimum funding should be €1 billion per year, additional to existing funding. This spending level should be reached rapidly, with the possibility to progressively increase it further, if appropriate, to bring the combined EU (Community, national and intergovernmental) security research investment level close to that of the U.S.
2. An ESRP should fund capability-related research projects up to the level of demonstrators that are useful in particular for Internal Security in the EU and for CFSP/ESDP-missions.
3. In closing the gap between civil and defence research, an ESRP should seek to maximize the benefits of multi-purpose aspects of technology. In order to stimulate synergies, it should encourage transformation, integration of applications and technology transfer from one sector to the other.
4. An ESRP should focus on interoperability and connectivity as key elements of cross-border and inter-service cooperation. In this context, a kernel of architectural design rules and standards should be worked out at an early stage.
5. The rules governing an ESRP must suit the specificities of security research. The Commission should, in consultation with all relevant stakeholders, develop the necessary rules for IPR and technology transfer.
6. Recognizing that many requirements will be government-specified, new financing instruments should be created to enable research funding to be disbursed, if justified, at up to 100% of cost.
7. A 'Security Research Advisory Board' should be established to draw strategic lines of action to prepare the research agenda of an ESRP as well as to advise on the principles and mechanisms for its implementation. Moreover, it should identify critical technology areas where Europe should aim for an indigenous competitive capability. The Board should consist of high-level experts from public and private customers, industry, research organizations and any other relevant stakeholders.
8. Definition of customer needs will be key for the successful implementation of an ESRP. A mechanism should therefore be established at EU level to identify in consultation with potential customers, future capability needs for Internal Security missions.
9. Effective coordination must make sure that the ESRP does not duplicate but complements other European research activities whether funded at Community, national or intergovernmental level.
10. The Commission and the Council should ensure an effective and efficient liaison between an ESRP and the future 'Agency in the field of defence capabilities development, research, acquisition and armaments'.

11. The ESRP should take into account and, where appropriate, coordinate with research efforts of international organizations with responsibilities for global or regional security issues.
12. An ESRP should aim at fostering the competitiveness of the European security industries and stimulating the development of the market (public and private) for security products and systems. Implementing the Proposals for Action put forward in the Commission's Communication 'Towards a European defence equipment market' would greatly help to achieve this objective and to maximize the benefits of an ESRP.

Based on this report, the EC chose to highlights the following four domains, through http://europa.eu.int/eur-lex/en/com/cnc/2004/com2004_0590en01.pdf:

1. Consultation and co-operation among all stakeholders through a 'European Security Research Advisory Board' (ESRAB, which is now being set up)
2. Development of a European Security Research Programme (ESRP) as a part of the 7th EU Research Framework Programme, to commence in 2007
3. Creation of an effective institutional framework that takes into account the Union's relevant policies, namely the Common Foreign and Security Policy (CFSP), European Security and Defence Policy (ESDP), and the new European Defence Agency (EDA)
4. Specific measures for the allocation of contracts and funding in security research

The next step was then the identification of the elements of the PASR 2005:

- Optimising security and protection of networked systems
- Protecting against terrorism (including bio-terrorism and incidents with biological, chemical and other substances)
- Enhancing crisis management (including evacuation, search and rescue operations, control and remediation)
- Achieving interoperability and integration of systems for information and communication
- Improving situation awareness (e.g. in crisis management, anti-terrorism activities, or border control)

3.2 The Norwegian Efforts Towards Security Research

In response to EC's plans on a security research programme, the Research Council of Norway (RCN) newly launched the "insight" project UTSIKT (Development Possibilities and Choice of Strategy within ICT) and its first outcome, in terms of the research programme VERDIKT (Core Competence and Added Value within ICT). A significant difference between RCN's initiatives UTSIKT and VERDIKT compared to the previous efforts within the ICT domain is the considerably increased focus on multi-disciplinary and multi-sectoral aspects of ICT research [8].

At the same time, an increased focus on security in the context of safety and the related risks is to observe within current sector-oriented research programmes, especially towards the transport and oil/energy sectors. The following gathers important conclusions drawn from the strategy and the topics of the funded projects related to the transport sector, in terms of the corresponding research programme RISIT:

1. There is a shift from the traditional deductive manner of research to a more holistic form, demonstrating more awareness about the potentials of a multidisciplinary research on particularly transport safety and security and related risk analysis.
2. The focus on experienced risk, as opposed to calculated (or "objective") risk is growing, among others, as a result of the shift explained above.
3. There is a new view on risk analysis; no longer as a single activity, but as a dynamic process that includes defining risk indicators as a function of both scenario-based data (involving analysis of future tendencies) and historical data, providing better models for risk communications, and more clarified representations of risk acceptance or rejection criteria. Additionally, this risk analysis process is now advocated for becoming an integrated part of the entire development process, including planning, construction and deployment of the systems/infrastructures subject to risk.
4. There is an increasing focus on decision analysis, which includes analysis and assessment of other alternative solutions for handling risks, as a decision's elements are usually based on the underlying risk analysis process, its resulting risk indicators and the suggested risk elimination, mitigation and containment/control mechanisms.
5. There is a growing concern about the risk compensation mechanisms and their consequences. These are basically caused by the unintended result from the implementation of safety countermeasures, namely the increased conviction and feeling of being safe amongst the public, leading many to take risks that are evaluated to be unacceptable by the same safety countermeasures.
6. In spite of the rapidly growing application of ICT systems for building and modernising the Norwegian transport infrastructures, the focus on research towards analysing the role of ICT systems and/or assuring their security has so far been negligible within the RISIT programme. On the basis of the recent granting of funds to several projects, however, there is a clear indication that this trend is about to change.

Finally, the RCN is planning to launch a new programme on Society Security (SAMRISK), where the multi-disciplinary and multi-sectoral nature of the research topics subject to granting funds is an absolute requirement for participation.

4 Conclusions

This paper has advocated the importance of a multi-disciplinary and multi-sectoral security research, and highlighted the European and Norwegian initiatives in that direction. The rationale for such a research is covered by addressing the recognised problems and ways of solutions from various technological and non-technological research fields and the corresponding customers from industrial, sociological, financial, political, defence-political, jurisprudential and environmental sectors. In that regard, some issues for further exploitation are brought to light. Next, the paper provides valuable information about the European and Norwegian initiatives towards security research of the nature stressed in the paper. For the prospective applicants and partners, the information would otherwise have been difficult to gather and use, as it is spread over numerous sites and documents.

References

1. NIST: Computer Security, Underlying Technical Models for Information Technology Security, http://csrc.nist.gov/publications/nistpubs/800-33/sp800-33.pdf.
2. EVANS, S., Heinbuch D., Kyle, E., Wallner, J.: “Agency Risk-Based Systems Security Engineering: Stopping Attacks with Intention”, IEEE Security & Privacy Transactions, November/December 2004, pp 59-62.
3. Yan, J., Blackwell, A., Anderson, R., Grant, A.: “Password Memorability and Security: Empirical Results”, IEEE Security & Privacy Transactions, September/October 2004, pp 25-31.
4. Thunem, A. P-J.: “Modelling of Knowledge Intensive Computerised Systems Based on Capability-Oriented Agent Theory (COAT)”, In Proc. *International IEEE Conference on Integration of Knowledge Intensive Multi-Agent Systems, IEEE-KIMAS’03*, pp 58-63, Cambridge (MA), USA, 2003.
5. International Atomic Energy Agency: “Planning and Preparing for Emergency Response to Transport Accidents Involving Radioactive Material”, Safety Guide, Safety Standard Series, No. TS-G-1.2 (ST-3).
6. International Standardisation Organisation: “Banking and related financial services (standards)”, ISO TC68.
7. http://www.cordis.lu/security/
8. http://www.forskningsradet.no/

Problem Frames and Architectures for Security Problems

Denis Hatebur[1] and Maritta Heisel[2]

[1] Universität Duisburg-Essen and Institut für technische Systeme GmbH
`denis.hatebur@uni-duisburg-essen.de`, `d.hatebur@itesys.de`
[2] Universität Duisburg-Essen, Germany, Fachbereich Ingenieurwissenschaften
`maritta.heisel@uni-duisburg-essen.de`

Abstract. We present several problem frames that serve to structure, characterize and analyze software development problems in the area of software and system security. These problem frames constitute *patterns* for representing security problems, variants of which occur frequently in practice. Solving such problems starts with the development of an appropriate software architecture. To support that process, we furthermore present architectural patterns associated with the problem frames. We illustrate our approach by the example of an electronic purse card.

1 Introduction

Problem frames were developed by Michael Jackson [6]. He describes them as follows (emphasis ours): "A problem frame is a kind of *pattern*. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."

Patterns are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure. Patterns are defined for different activities at different stages of the software life cycle. *Problem Frames* [6] are patterns that classify software development *problems*. *Architectural styles* are patterns that characterize software architectures [1, 11]. They are also called "architectural patterns" (see Section 2.2). *Design Patterns* [5] are used for finer-grained software design[1], while *idioms* are low-level patterns related to specific programming languages [3].

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time. The problem frames defined by Jackson cover a large number of software development problems, because they are quite general in nature. To support software development in more specific areas, however, specialized problem frames are needed.

In this paper, we present four problem frames that capture software development problems occurring frequently in the area of software and system security. We call these problem frames *security frames*. Two of our security frames concern authentication. The third one deals with the secure (i.e., encrypted) transmission of data, and the fourth one

[1] Design patterns for security have also been defined, see Section 5.

is suitable for generating and storing security information (such as public and private keys, PINs).

Architectural patterns are suitable solution structures for problem frames, because architectural design is one of the first activities in solving software development problems. Hence, the gap between the problem description and the software architecture is not too large, and we can establish direct relations between problem structures and solution structures. As we have shown in [4], one can define architectural patterns that reflect the characteristics of the different problem frames. In much the same way, we equip our security frames with corresponding architectural patterns.

Section 2 describes the basics of our work, while the security frames and corresponding architectures are presented in Section 3. We illustrate our approach by developing a secure electronic purse card in Section 4. Section 5 discusses related work, and we conclude in Section 6.

2 Problem Frames and Architectural Patterns

In this paper, we present new problem frames for security problems and the corresponding architectural patterns. As a notation for our architectural patterns, we use composite structure diagrams of UML 2.0 [12]. In the following, we give brief descriptions of these basic concepts of our work.

2.1 Problem Frames

Problem frames are described by *frame diagrams*, which basically consist of rectangles and links between these, see left-hand side of Fig. 1. The task is to construct a *machine* that improves the behavior of the environment it is integrated in.

Plain rectangles denote application domains (that already exist), a rectangle with a double vertical stripe denotes the machine to be developed, and requirements are denoted with a dashed oval. The connecting lines represent interfaces that consist of shared *phenomena*. A dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

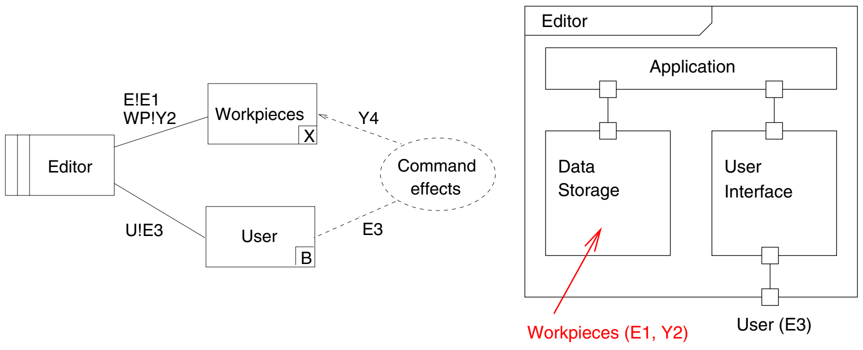


Fig. 1. Workpieces Frame Diagram and Architectural Pattern

Jackson distinguishes *causal* domains that comply with some laws, *lexical* domains that are data representations, and *biddable* domains that are usually people. Jackson defines five basic problem frames (*Required Behaviour, Commanded Behaviour, Information Display, Workpieces* and *Transformation*). As an example, we present the *Workpieces* frame in more detail. The following problems fit to that problem frame [6]: "A tool is needed to allow a user to create and edit a certain class of computer processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analyzed or used in other ways. The problem is to build a machine that can act as this tool." The "X" indicates that the *Workpieces* domain of the frame diagram shown on the left-hand side of Fig. 1 is a lexical domain. The notation "U!E3" means that the user commands *E3* are controlled by the (biddable) *User* domain. Similarly, the phenomena *E1* are the commands used by the *Editor* to change the *Workpieces* domain. The shared phenomena *Y2* represent the state of a workpiece; they are controlled by the *Workpieces* domain. The shared phenomena *Y4* need not be the same as *Y2*. They will often have some meaning to the user, whereas the phenomena *Y2* are phenomena accessible by the machine.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements, and it is not shown who is in control of the shared phenomena. An example of a context diagram is shown in Fig. 8. Then, the problem is decomposed into subproblems. If ever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, phenomena, interfaces and requirements. The instantiated frame diagram is called a *problem diagram* (for an example, see Fig. 9). It describes the problem as a whole. Since the requirements refer to the environment in which the machine must operate, the next step consists in deriving a *specification* for the machine, using domain knowledge. In that process, non-implementable requirements are transformed into implementable ones. (For a more detailed description, see [7].) The specification is the starting point for the development of the machine.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. Since all problems fitting in a problem frame share the same characteristic properties, their solutions will have common characteristic properties, too. Therefore, it is worthwhile to look for solution structures that match the problem structures defined by problem frames.

2.2 Architectural Styles

According to Bass, Clements, and Kazman [1], "the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." Architectural styles are patterns for software architectures.

When choosing an architecture for a system, usually several architectural styles are possible. However, instead of considering *all* possible architectures, we propose *specific* architectural patterns for our security frames in order to provide a concrete starting point for the further development of the machine. The architectural patterns we have defined for Jackson's problem frames (see [4]) and the ones we will define for security frames are based on a *layered architecture*. The components in this layered architecture are either *communicating processes* (active components), or they are used with a *call-and-return* mechanism (passive components). That design decision is taken in a later step of the development. In [4], we also show how the *repository* and the *pipe-and-filter* architectural styles can be integrated into the layered architecture. We use UML 2.0 composite structure diagrams (see Section 2.3) to represent architectural patterns as well as concrete architectures.

The architectural pattern shown on the right-hand side of Fig. 1 contains a user interface component, because the problem frame diagram contains a user. The data storage component of the architecture corresponds to the *Workpieces* domain of the frame diagram. The *Editor Application* component is responsible for manipulating the data storage according to the user commands. Note that there is only one interface with the environment – namely the interface with the user – because the lexical *Workpieces* domain is part of the machine.

2.3 Composite Structure Diagrams

Composite structure diagrams [12] are a means to describe architectures. They contain named rectangles, called *parts*. These parts are components of the software. Each component may contain other (sub-) components. Atomic components can be described by state machines and operations for accessing internal data. In our architectures, components for data storage are only included if the data are stored persistently. Otherwise they are assumed to be part of some other component. Parts may have *ports*, denoted by small rectangles. Ports may have interfaces associated to them. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation.

Fig. 2 shows how interfaces in problem diagrams are transformed into interfaces in composite structure diagrams. The partial problem diagram shown on the left-hand side of Fig. 2 states that the phenomena *phen1* and *phen2* shared between the machine and a domain are controlled by the machine. In the composite structure diagram (with associated interface class) shown in the middle of Fig. 2, this is expressed by a required interface *P1_if* of the *part* component of the machine, which is the same as for the whole machine. Shared phenomena controlled by a domain correspond to provided instead of

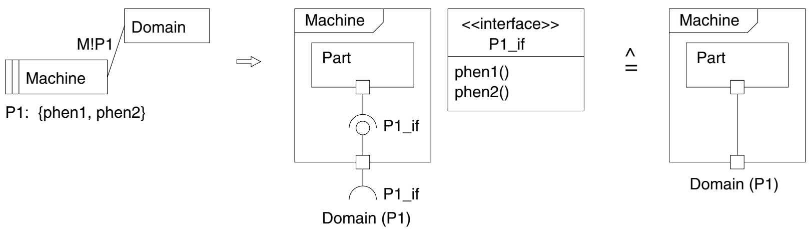


Fig. 2. Notation for Architectures

required interfaces of the *part* and the machine, respectively. Because of this direct correspondence, we do not use the socket and lollipop notation in the following, but use connectors between ports, as shown on the right-hand side of Fig. 2. These connectors can be implemented e.g. as data streams, function calls, asynchronous messages or hardware access.

3 Security Frames and Architectural Patterns

We now present the four security frames we have developed, together with the corresponding architectural patterns that define structures for the machine domains of the security frames.

The first two security frames are concerned with authentication. We distinguish two authentication frames. In the first frame, a subject must authenticate itself to the machine to be constructed. In the second frame, the machine to be constructed must authenticate itself to some other subject. The third security frame deals with the secure transmission of data over an insecure channel, and the fourth frame is applicable when common security knowledge must be distributed with the help of a trust center. None of these problem classes is addressed by Jackson's problem frames.

3.1 Accept Authentication Frame

For security systems, authentication of users and other components is an important concern. Authentication is necessary to allow access to some other information. That information is not part of the problem and hence not part of the frame diagram shown on the left-hand side of Fig. 3.

The *Subject* in the frame diagram can be a user or another machine. To make authentication possible, there must be a common knowledge between subject to be authenticated and the machine. If the authentication information *S2* provided by the subject matches the common knowledge *S1* stored in the machine, then the authentication is successful. Otherwise, it fails. That information is represented by the domain *Security state*.

In the corresponding architectural pattern on the right-hand side of Fig. 3, we include the *Common knowledge*, but we do not include the *Security state* because it should not be stored persistently and hence does not correspond to an architectural component.

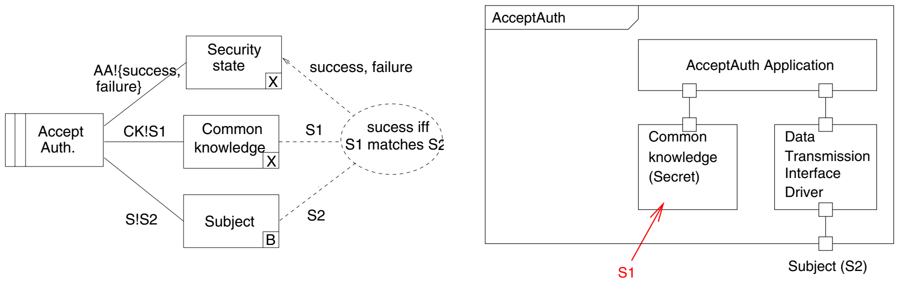


Fig. 3. Accept Authentication Information Frame Diagram and Architectural Pattern

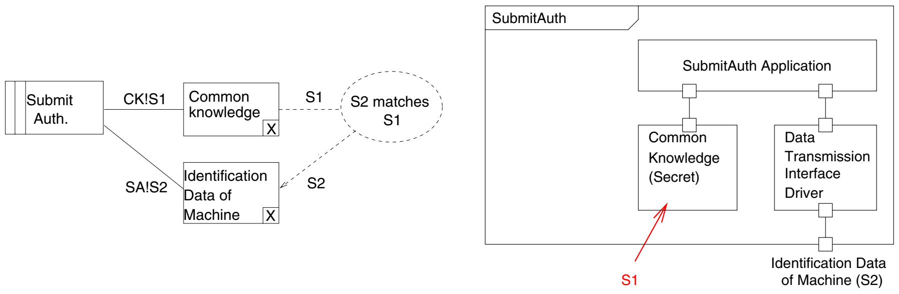

Submit Auth.
CK!S1
Common knowledge
X
S1
S2 matches S1
SA!S2
Identification Data of Machine
X
S2
SubmitAuth
SubmitAuth Application
Common Knowledge (Secret)
Data Transmission Interface Driver
S1
Identification Data of Machine (S2)

Fig. 4. Submit Authentication Information Frame Diagram and Architectural Pattern

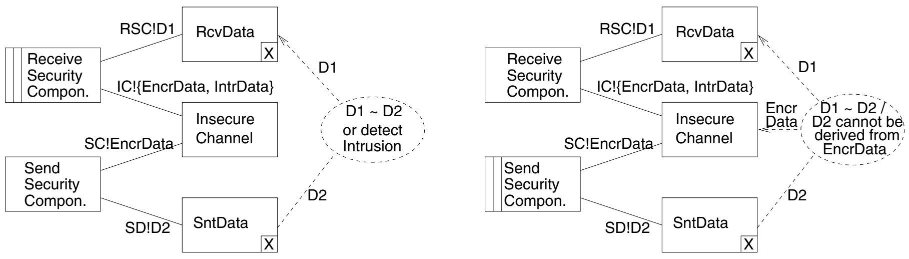


Fig. 5. Secure Data Transmission Frame Diagrams

Instead, it is reflected in the internal state of the part *AcceptAuth Application* that is responsible for enabling or disabling other functionality.

3.2 Submit Authentication Frame

Because the subject might be a system, there exists the problem *Submit Authentication*. The frame diagram and the corresponding architectural pattern are shown in Fig. 4. For this problem, the *Security state* is part of the subject to which the machine to be built wants to authenticate itself. Therefore the *Security State* it is not part of the frame diagram. The machine has to use the *Common knowledge* to generate matching *Identification Data* for the subject.

3.3 Secure Data Transmission Frames

Another important security problem is the secure transmission of data. We need to build a security component that receives data from another component or sends data to another component over an insecure channel. That situation is depicted in Fig. 5 on the left-hand side for receiving data and on the right-hand side for sending data.

The security component at the bottom of the figure wants to send data (domain *SntData*, phenomena *D2*) to the security component at the top of the figure. Because the transmission channel is insecure, the data is encrypted (phenomena *EncrData*). It is possible that the insecure channel transmits some intruder data *IntrData* instead

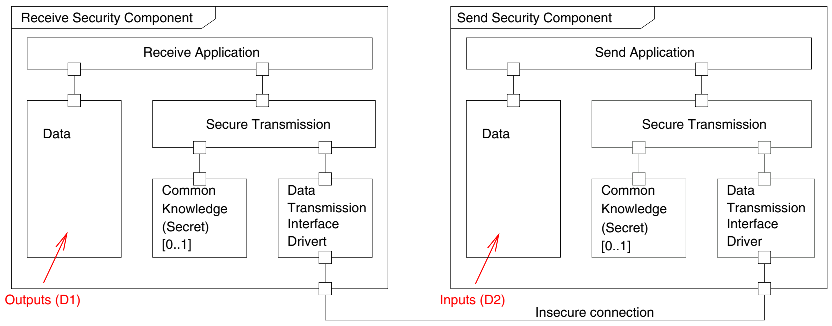

Receive Security Component
Receive Application
Data
Secure Transmission
Common Knowledge (Secret) [0..1]
Data Transmission Interface Drivert
Outputs (D1)
Send Security Component
Send Application
Data
Secure Transmission
Common Knowledge (Secret) [0..1]
Data Transmission Interface Driver
Inputs (D2)
Insecure connection

Fig. 6. Secure Data Transmission Architectural Pattern

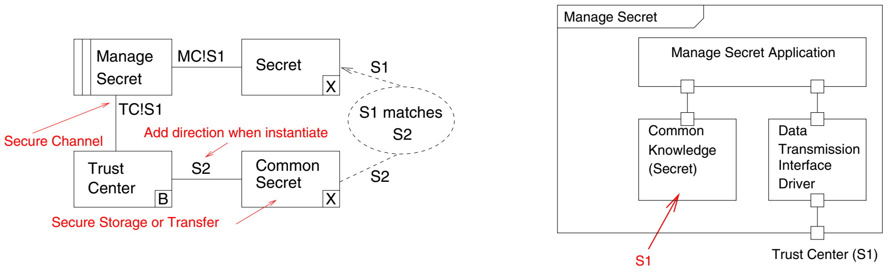

Manage Secret
MC!S1
Secret
S1
TC!S1
S1 matches S2
Secure Channel
Add direction when instantiate
Trust Center
B
S2
Common Secret
X
S2
Secure Storage or Transfer
Manage Secret
Manage Secret Application
Common Knowledge (Secret)
Data Transmission Interface Driver
S1
Trust Center (S1)

Fig. 7. Distribute Security Information Frame Diagram and Architectural Pattern

of the original encrypted data. The encrypted data or intruder data will be decrypted by the security component shown on the top of the figure, yielding *D1*. The requirement for receiving data states that either the data *D1* and *D2* match, or the intrusion will be detected (integrity). The requirement for sending data states that the data *D1* and *D2* match, and that the *D1* cannot be derived from *EncrData* (integrity and confidentiality).

For this class of problems, we propose the architectural patterns shown in Fig. 6. In this architecture, a storage for a secret is necessary in addition to the data storage. If this storage is persistent it is an additional part in the architecture. Otherwise, the storage component is not included, as indicated by the notation [0..1].

3.4 Distribute Security Information Frame

For the architecture shown in Fig. 6, it is necessary that each machine has some common knowledge. This raises the problem of how to distribute that common knowledge. Fig. 7 shows the frame diagram. The common (secret) knowledge is transferred to the machine by a trusted component, the *Trust Center*, over a secure channel. The requirement states that indeed the correct common knowledge is stored in the machine.

The corresponding architectural pattern contains a part *ManageSecretApplication* that has to store the secret (or common knowledge) and restrict the access to it. Its purpose is to manage the secret.

4 Case Study: Electronic Purse Card

The illustrate the usage of security frames, we consider a smart card with a simplified electronic purse application using asymmetric encryption. This smart card is used to ensure secure payment. To pay with the card, the user has to enter a PIN at a card reader. The authorization of the card is checked via a website. The card also has to check the authorization of the website. The transmitted data have to be protected against unauthorized read and change access. To allow payment, money must be loaded on the card. This is only possible if the the account information allows this transaction (the card can be locked).

4.1 Requirements and Context Diagram

The following requirements must be met. We number them in order to reference them in the description of the different subproblems.

R1 Loading money on the card is possible if the account information allows to do this transaction.
R2 Paying with the card is possible if there is enough money on the card.
R3 Authentication of the card is necessary for paying and loading money.
R4 Authentication of the website is necessary for paying and loading money.
R5 Authentication of the user using a PIN is necessary for paying.
R6 The card should prevent replay-intrusion and even prevent somebody else from reading transmitted information (man-in-the-middle attack).
R7 It should not be possible to copy the card.
R8 Only a card and a website personalized by a trust center should be usable for transactions.

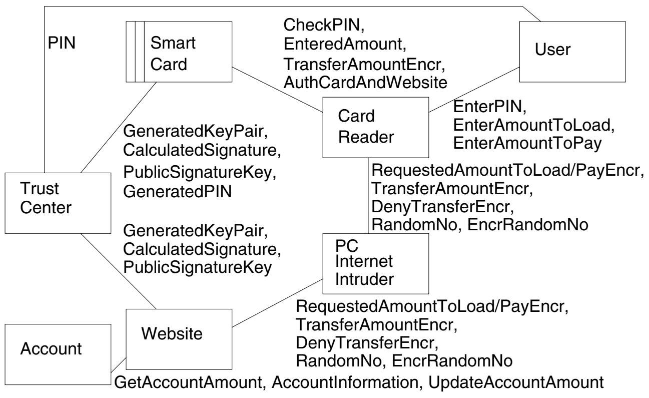

PIN
Smart Card
CheckPIN, EnteredAmount, TransferAmountEncr, AuthCardAndWebsite
User
Card Reader
EnterPIN, EnterAmountToLoad, EnterAmountToPay
GeneratedKeyPair, CalculatedSignature, PublicSignatureKey, GeneratedPIN
Trust Center
RequestedAmountToLoad/PayEncr, TransferAmountEncr, DenyTransferEncr, RandomNo, EncrRandomNo
GeneratedKeyPair, CalculatedSignature, PublicSignatureKey
PC Internet Intruder
RequestedAmountToLoad/PayEncr, TransferAmountEncr, DenyTransferEncr, RandomNo, EncrRandomNo
Account
Website
GetAccountAmount, AccountInformation, UpdateAccountAmount

Fig. 8. Context diagram for Electronic Purse Card

Fig. 8 shows the context diagram corresponding to this problem. It contains the relevant domains and shared phenomena. The domains *SmartCard*, *Website* and *Account* occur only once in the diagram. However, the system will work with different instances of these domains. It will be able to handle different smart cards, and it will be connected via different websites to different accounts. Moreover, the interface between the *CardReader* and the *User* has been simplified. The domain *PC, Internet, Intruder* denotes the insecure channel that involves a PC, the Internet, and possibly an intruder.

The following table shows the subproblems that can be identified, the problem/ security frame the subproblem fits to, and the requirements that are covered. In the following, we present one instantiation for each of the introduced problem/security frames.

Subproblem	Frame	Reqs.
Load Money	Workpieces	R1
Pay	Workpieces	R2
Authenticate Card	Submit Authentication	R3
Authenticate Website	Accept Authentication	R4
PIN Authentication	Accept Authentication	R5
Receive Secure Data	Secure Data Transmission	R6
Send Secure Data	Secure Data Transmission	R6
Distribute Keys	Distribute Security Information	R7, R8
Distribute PIN	Distribute Security Information	R7, R8

4.2 Subproblem: Load Money

This subproblem is concerned with loading money on the card (R1). It fits to a variant of Jackson's *Workpieces* problem frame. It is extended with the constraint that only if the account information allows it, the amount of money can be loaded onto card. The problem shown in Fig. 9 states that the *CardAmount* should change according to *EnterAmountToLoad* and *AccountInformation*.

The problem diagram of Fig. 9 is derived from the context diagram of Fig. 8 as follows: the domain *Trust Center* is not relevant for this subproblem. The connection domains[2] *Card Reader* and *PC, Internet, Intruder* are left out in this subproblem, because the connection is assumed to be secure, the security of the connection being covered by other subproblems. To describe this problem, we split the domain *SmartCard* into *Money on Card* and *Load Money*.

The problem of Fig. 9 shows the requirements the machine must achieve when integrated into its environment. As noted earlier, the requirements must be transformed into a specification that describes the behavior of the machine. In the area of security, protocols [9] exist that make it possible to transform requirements such as "secure transmission" of "authentication" into sequences of messages exchanged between different partners.

Fig. 10 shows a UML 2.0 sequence diagram that represents the specification of the machine *Load Money*. When the *User* enters the amount of money (*EnterAmountToLoad*) the message *RequestedAmountToLoadEncr* will be sent to the *Website*. The *Website* will check the *AccountInformation*. The sequence diagram in Fig. 10 describes

[2] These are domains that serve to connect two other domains. If a connection domain is reliable and does not cause significant delays, it may be ignored, see [6].

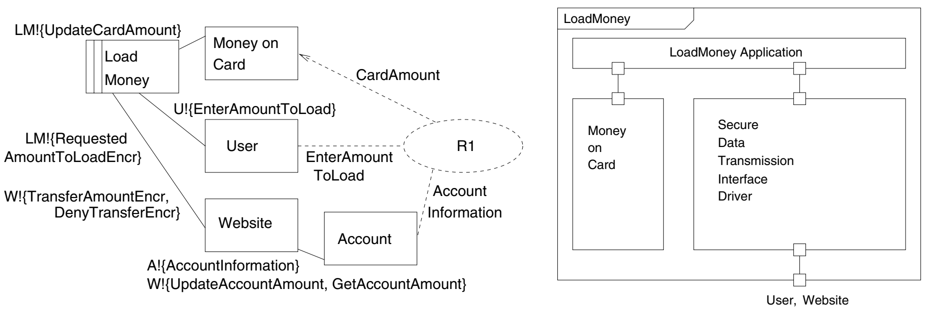


Fig. 9. Load Money Subproblem Diagram and Architecture

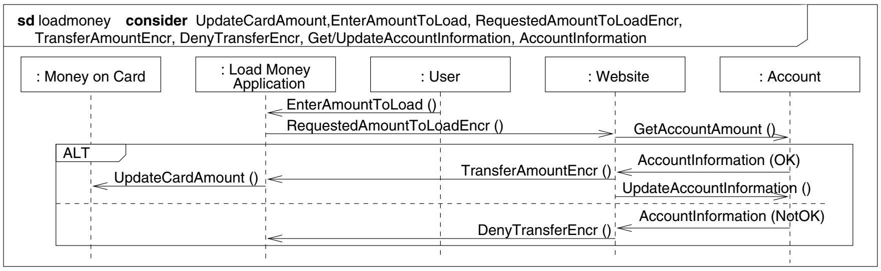

sd loadmoney consider UpdateCardAmount,EnterAmountToLoad, RequestedAmountToLoadEncr, TransferAmountEncr, DenyTransferEncr, Get/UpdateAccountInformation, AccountInformation
: Money on Card
: Load Money Application
: User
: Website
: Account
EnterAmountToLoad ()
RequestedAmountToLoadEncr ()
GetAccountAmount ()
ALT
AccountInformation (OK)
TransferAmountEncr ()
UpdateCardAmount ()
UpdateAccountInformation ()
AccountInformation (NotOK)
DenyTransferEncr ()

Fig. 10. Load Money Sequence Diagram

the following two alternatives, marked with the keyword *ALT*. If the *AccountInformation* allows to load the requested amount of money on card, the amount of money on the *Account* will be updated (*UpdateAccountInformation*), the amount will be transmitted (*TransferAmountEncr*), and *Money on Card* will be updated (*UpdateCardAmount*[3]). Otherwise the phenomenon *DenyTransferEncr* occurs. For reasons of space, we do not give the sequence diagrams for the other subproblems. For this subproblem, the *Website* is the website of the user's bank, and the *Account* is the user's account, which is debited with the account loaded onto the card.

The right-hand side of Fig. 9 shows the corresponding architecture, which is an instantiation of the pattern given in Fig. 1. Here, *User* and *Website* are connected to the machine via a *Secure Data Transmission Interface*.

4.3 Subproblem: Authenticate Website

An authentication of the website is required in R4. Here, we instantiate the "Accept Authentication" frame as shown in Fig. 11. For this authentication, a *Random Number* should be used to prevent replay intrusion. Therefore, we need to add the random number RN as a shared phenomenon between the *Auth Website* machine and *Website*, controlled by the machine.

[3] With the new domain *Money on Card* the shared phenomenon *UpdateCardAmount* has to be added.

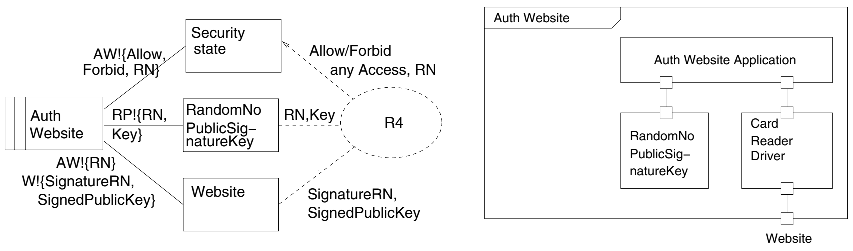

Security state
AW!{Allow, Forbid, RN}
Allow/Forbid any Access, RN
Auth Website
RP!{RN, Key}
RandomNo PublicSig–natureKey
RN,Key
R4
AW!{RN}
W!{SignatureRN, SignedPublicKey}
Website
SignatureRN, SignedPublicKey
Auth Website
Auth Website Application
RandomNo PublicSig–natureKey
Card Reader Driver
Website

Fig. 11. Website Authentication Subproblem Diagram and Architecture

The part *Auth Website* of the architecture shown in Fig. 11 must contain a part *RandomNo PublicSignatureKey* that can generate random numbers with sufficient quality. To check the authenticity of the website, the website encrypts the random number provided by the card with its private key (it generates the signature of the random number (*SignatureRN*). This signature can be checked using the public key of the website. To make it possible that new websites can be added to the system without replacing all cards, the website has to provide its own public key. The card can check the provided public key using a signature of a Trust Center. The interaction for a combination of submitting and accepting authentications can be found in [10].

4.4 Subproblem: Receive Secure Data

To prevent replay intrusion and read access on transmitted data, we define the subproblem shown in Fig. 12. The *Card Reader* and the *Trust Center* are not directly relevant for this subproblem. Moreover, is not relevant what data are transmitted. Therefore we take an abstraction from *Load Money* and *Pay*. Also the messages *TransmitAmountEncrMessage* and *CheckAmountEncrMessage* are merged to *AccessAmountEncr*.

A common secret as described in the architectural pattern of Fig. 6 is not stored persistently on the card. It can be derived from the random number and can be changed for each transmission. Hence, the architecture of Fig. 12 (derived from the pattern given in Fig. 6) does not contain a corresponding component.

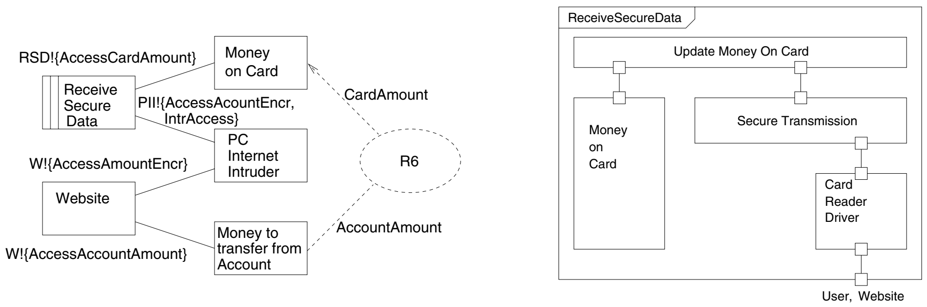


Fig. 12. Receive Secure Data Subproblem Diagram and Architecture

4.5 Subproblem: Distribute Keys

Requirements R7 and R8 express that only the trust center may generate a valid card. Important for a valid card are the PIN and the keys. In this subproblem, we focus on the keys. The requirements are covered partly in the subproblem shown in Fig. 13, where the *Common Secret* domain that is part of the *Trust Center* is shown separately. That subproblem is an instance of the "Distribute Security Information" frame.

The trust center has to generate an individual public/private key pair for each card and write it onto the card. To guarantee that this key pair is valid and originated from the trust center, it is signed with the private key of the trust center. To allow the card to authenticate other systems, it needs the public key of the trust center. This also is written onto the card.

The subproblem the machine *Manage Secrets* has to solve is to manage the access to the security information. After producing the card, everybody owning an uninitialized card can initialize it. But only the trust center is able to generate a signature with its private key that allows the key to be used in the payment system. The machine has to manage the access to the secrets. After initializing the card, the functionality to change the security information is disabled. The private part of the key pair must also be protected against read access. Moreover, all other functionality has to be disabled as long as the card is not yet initialized.

The architecture of Fig. 13 is an instantiation of the pattern given in Fig. 7.

4.6 Composed Architecture

We now must compose the architectures developed for the subproblems to obtain an architecture for the whole Smart Card. For doing this, we must find the parts occurring in different subproblem architectures that must be mapped to the same component in the composed architecture.

The composed architecture for the Smart Card is shown in Fig. 14. The component *Amount Of Money* combines the persistent storage of the subproblems *Pay* and *Load Money*. The *Load Money/Pay-Application* combines the behavior of the machines *Load Money* and *Pay*. The component *Secure Data Transmission Interface* is replaced by the components of the security subproblem architectures (including the ones given in

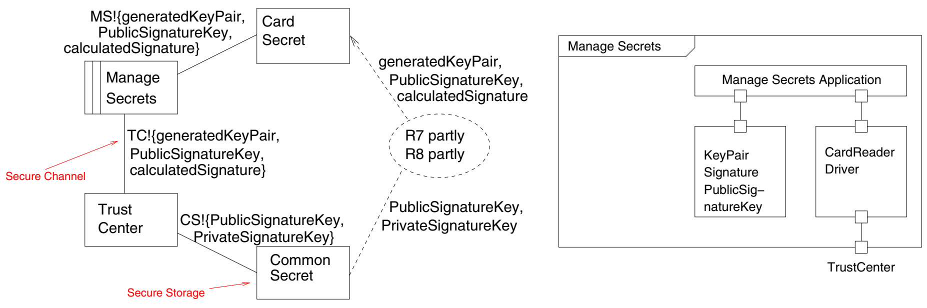
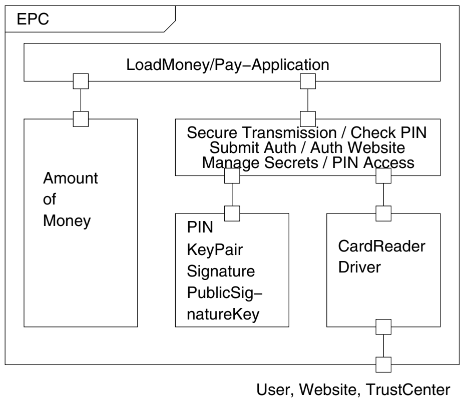

MS!{generatedKeyPair, PublicSignatureKey, calculatedSignature}
Card Secret
Manage Secrets
generatedKeyPair, PublicSignatureKey, calculatedSignature
TC!{generatedKeyPair, PublicSignatureKey, calculatedSignature}
R7 partly
R8 partly
Secure Channel
Trust Center
CS!{PublicSignatureKey, PrivateSignatureKey}
PublicSignatureKey, PrivateSignatureKey
Common Secret
Secure Storage
Manage Secrets
Manage Secrets Application
KeyPair Signature PublicSig-natureKey
CardReader Driver
TrustCenter

Fig. 13. Distribute Keys Subproblem Diagram and Architecture



EPC
LoadMoney/Pay–Application
Amount of Money
Secure Transmission / Check PIN
Submit Auth / Auth Website
Manage Secrets / PIN Access
PIN
KeyPair
Signature
PublicSig–natureKey
CardReader Driver
User, Website, TrustCenter

Fig. 14. Composed Architecture

Figs. 11–13). The *Card Reader Driver* is the same in all security subproblem architectures and can be used directly in the composed architecture. The functionality of the remaining two components have to be derived from those in the security subproblem architectures.

For all components, their exact specifications must be set up, and it must be shown that the components work together in such a way that they fulfill the specifications of all machines corresponding to the different subproblems. The functionalities of the different architectural parts are now clear, as well as the interfaces between them. Thus, we have established an appropriate starting point for the further development of the smart card system in a systematic way.

5 Related Work

Our security frames are related to abuse frames on the one hand and to security patterns on the other hand.

Security frames treat security requirements in the same way as other (functional) requirements, and the goal is to construct a machine that fulfills the security requirements. Lin et al. [8] take another approach to use the ideas underlying problem frames in security. They define so-called anti-requirements and the corresponding abuse frames. An anti-requirement expresses the intentions of a malicious user, and an abuse frame represents a security threat. The purpose of anti-requirements and abuse frames is to analyze security threats and derive security requirements. Thus, the two approaches complement each other. Abuse frames can be used to derive the security requirements that can then be addressed with security frames.

While abuse frames can be used earlier in the software development process than security frames, security patterns [2] are applied in a later phase, namely the phase of detailed design. The relation between security frames and security patterns is much the same as the relation between problem frames and design patterns: the frames describe problems, whereas the design/security patterns describe solutions on a fairly detailed level of abstraction. Moreover, design and security patterns are applicable only in an object-oriented setting, while problem and security frames are independent of a particular programming paradigm.

6 Conclusion

In this paper, we have presented a new kind of problem frames tailored for representing security problems, called security frames. Security frames are patterns for software development problems occurring frequently when security-critical software has to be developed.

The security frames presented in this paper are intended to be the first in a more complete collection. Once a (relatively) complete collection of security frames is defined, it is of considerable help for developers. For a new security-critical system to be constructed, the security frame catalogue can be inspected in order to find the frames that apply for the given problem. Thus, a security frame catalogue helps to avoid omissions and to cover all security aspects that are relevant for the given problem.

Furthermore, the security frames help to decompose complex security problems to simpler ones that can be handled by standard mechanisms. Like design and security patterns, security frames can establish a common vocabulary and shared knowledge between developers of security-critical systems.

While the security frames themselves “only” help to comprehend, locate and represent problems, our architectural patterns associated with the different security frames propose concrete structures for *solving* the problems fitted to security frames. The architectural patterns also help to compose the solutions of the different subproblems in order to construct the complete system, as is shown in more detail in [4].

With the concept of security frames and corresponding architectural patterns (in addition to abuse frames and security patterns), one can hope to cover large parts the development of security-critical software with a pattern-based approach.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] B. Blakley and C. Heath. *Technical Guide: Security Design Patterns*. The Open Group, April 2004. `http://www.opengroup.org/publications/catalog/g031.htm`.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] C. Choppy, D. Hatebur, and M. Heisel. Architectural patterns for problem frames. *IEE Proceedings – Software, Special issue on Relating Software Requirements and Architecture*, 2005. To appear.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[6] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[7] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[8] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Introducing abuse frames for analysing security requirements. In *Proceedings of 11th IEEE International Requirements Engineering Conference (RE'03)*, pages 371–372, 2003. Poster Paper.

[9] C. P. Pfleeger. *Security in Computing*. Prentice Hall, 1996.

[10] T. Rottke, D. Hatebur, M. Heisel, and M. Heiner. A problem-oriented approach to common criteria certification. In S. Anderson, S. Bologna, and M. Felici, editors, *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 2434, pages 334–346. Springer-Verlag, 2002.

[11] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[12] UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.

Author Index