

Friedemann Bitsch
J r mie Guiochet
Mohamed Ka nliche (Eds.)

LNCS 8153

Computer Safety, Reliability, and Security

32nd International Conference, SAFECOMP 2013
Toulouse, France, September 2013
Proceedings



 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Friedemann Bitsch Jérémie Guiochet
Mohamed Kaâniche (Eds.)

Computer Safety, Reliability, and Security

32nd International Conference, SAFECOMP 2013
Toulouse, France, September 24-27, 2013
Proceedings



Springer

Volume Editors

Friedemann Bitsch
Thales Transportation Systems GmbH
Lorenzstraße 10, 70435 Stuttgart, Germany
E-mail: friedemann.bitsch@thalesgroup.com

Jérémie Guiochet
Mohamed Kaâniche
University of Toulouse
Laboratory of Analysis and Architecture of Systems (LAAS-CNRS)
Dependable Computing and Fault Tolerance Group
7 avenue du Colonel Roche, 31031 Toulouse, France
E-mail: {jeremie.guiochet, mohamed.kaaniche}@laas.fr

ISSN 0302-9743
ISBN 978-3-642-40792-5
DOI 10.1007/978-3-642-40793-2
Springer Heidelberg New York Dordrecht London

e-ISSN 1611-3349
e-ISBN 978-3-642-40793-2

Library of Congress Control Number: 2013946883

CR Subject Classification (1998): K.6.5, C.2, D.2, H.3, D.4.6, E.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

On behalf of the Technical Program Committee members, we welcome you to the proceedings of the 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013), held in Toulouse, France.

SAFECOMP is a major event that provides a forum for researchers and practitioners from all over the world to present and discuss their latest research results and experiments on the development, assessment, operation, and maintenance of safety-related and safety-critical computer systems. Since it was established in 1979 by the European Workshop on Industrial Computer Systems, Technical Committee 7 on Reliability, Safety and Security (EWICS TC7), SAFECOMP has contributed to the progress of the state of the art in dependability of computers in safety-related and safety-critical systems. SAFECOMP provides ample opportunity to exchange insights and experience on emerging methods, approaches, and practical solutions. It is a one-stream conference without parallel sessions, allowing easy networking.

This year, we received 88 submissions from 24 countries. The review and selection process followed the tradition of thoroughness and rigor of SAFECOMP. The review process was organized in two phases, mainly relying on the Program Committee (PC) members for the evaluation of the papers. In the first phase, each submitted paper was assigned to three members of the PC, which was composed of 51 researchers and industrials from 16 countries (Europe, North and South America, and Asia). At the end of this phase, a PC meeting was held in Zürich attended by 29 PC members. The selection process was competitive with a 28% acceptance rate.

The 25 papers selected in the program include 20 regular papers and five practical experience reports. This year's program was organized into 11 sessions, covering different assessment methods (testing and verification, software reliability, failure mode analysis, safety assurance) and addressing a wide variety of interesting topics (security, error control code, dependable user interfaces). The technical program was complemented by keynote presentations from three distinguished speakers. John Rushby (SRI, USA) gave a talk on logic and epistemology in safety cases, Pascale Traverse (Airbus) addressed the dependability in embedded systems considering the airbus fly-by-wire system as an example, and finally Sami Haddadin (DLR, Germany) addressed safe human-robot physical interaction. Additionally, a session for fast abstract presentation was included in the program to promote interactions on novel ideas or work in progress, or opinion pieces that can address any issue relevant to SAFECOMP topics of interest.

We would like to express our deep gratitude to the PC members, who devoted their time and provided their expertise to ensure the quality of the reviewing and the shepherding processes. We are also very grateful to the external reviewers

for their outstanding help and assistance in the review process. Needless to add that the elaboration of the SAFECOMP 2013 technical program would not have been possible without the greatly appreciated contribution of all the authors who submitted their work to the conference.

The support received from the EWICS TC7 committee chaired by Francesca Saglietti is very much appreciated. Special thanks go to Karama Kanoun for her continuous support and feedback during the organization of this edition, and to Frank Ortmeier for his help based on his experience as SAFECOMP general chair and PC chair in 2012. We would also like to thank Friedemann Bitsch, Matthieu Roy, Marc-Olivier Killijian, Nicolas Rivière, and Pascal Traverse who were publication, workshops, fast abstracts, publicity, and industry liaison chairs, respectively. Also special thanks to Christoph Schmitz, who hosted the PC meeting in Zürich, Zühlke Engineering AG. Finally, we would like to express our gratitude to Sonia de Sousa, Yves Crouzet, members of the TSF research group, the LAAS-CNRS administrative and technical staff, and the sponsors for their assistance in organizing this event.

We hope that you will benefit from the conference proceedings.

Jérémie Guiochet
Mohamed Kaâniche

Organization

Organizing Committee

General Chair

Jérémie Guiochet LAAS-CNRS, University of Toulouse, France

Program Co-Chairs

Mohamed Kaâniche LAAS-CNRS, University of Toulouse, France

Jérémie Guiochet LAAS-CNRS, University of Toulouse, France

EWICS TC7 Chair

Francesca Saglietti University of Erlangen-Nuremberg, Germany

Publication Chair

Friedemann Bitsch Thales Transportation Systems GmbH, Germany

Local Organizing Committee

Yves Crouzet LAAS-CNRS, University of Toulouse, France

Sonia De Sousa LAAS-CNRS, University of Toulouse, France

Fast Abstract Chair

Marc-Olivier Killijian LAAS-CNRS, University of Toulouse, France

Workshop Chair

Matthieu Roy LAAS-CNRS, University of Toulouse, France

Publicity Chair

Nicolas Rivière LAAS-CNRS, University of Toulouse, France

Industry Liaison Chair

Pascal Traverse EADS-Airbus, Toulouse, France

International Program Committee

Anderson, Stuart (UK)

Blanquart, Jean-Paul (France)

Bieber, Pierre (France)

Bloomfield, Robin (UK)

Bitsch, Friedemann (Germany)

Bologna, Sandro (Italy)

Bondavalli, Andrea (Italy)
Braband, Jens (Germany)
Casimiro, Antonio (Portugal)
Cotroneo, Domenico (Italy)
Cukier, Michel (USA)
Daniel, Peter (UK)
Denney, Ewen W. (USA)
Ehrenberger, Wolfgang (Germany)
Felici, Massimo (UK)
Flammini, Francesco (Italy)
Gorski, Janusz (Poland)
Grunske, Lars (Germany)
Guiochet, Jérémie (France)
Halang, Wolfgang (Germany)
Heisel, Maritta (Germany)
Johnson, Chris (UK)
Jonsson, Erland (Sweden)
Kaâniche, Mohamed (France)
Kanoun, Karama (France)
Karlsson, Johan (Sweden)
Kelly, Tim (UK)
Knight, John (USA)
Koopman, Phil (USA)

Koornneef, Floor (The Netherlands)
Lindskov Hansen, Søren (Denmark)
Moraes, Regina (Brazil)
Motet, Gilles (France)
Nanya, Takashi (Japan)
Nordland, Odd (Norway)
Ortmeir, Frank (Germany)
Palanque, Philippe (France)
Paulitsch, Michael (Germany)
Romano, Luigi (Italy)
Romanovsky,
 Alexander (UK)
Rugina, Ana (The Netherlands)
Saglietti, Francesca (Germany)
Schmitz, Christoph (Switzerland)
Schoitsch, Erwin (Austria)
Skavhaug, Amund (Norway)
Steininger, Andreas (Austria)
Sujan, Mark (UK)
Trapp, Mario (Germany)
Troubitsyna, Elena (Finland)
Van der Meulen, Meine (Norway)
Weaselynck, Hélène (France)

External Reviewers

Alebrahim, Azadeh (Germany)
Andrews, Zoe (UK)
Beckers, Kristian (Germany)
Brancati, Francesco (Italy)
Ceccarelli, Andrea (Italy)
Cicotti, Giuseppe (Italy)
Cinque, Marcello (Italy)
Clarotti, Carlo (Italy)
Conmy, Philippa (UK)
Craveiro, Joao Pedro (Portugal)
Di Sarno, Cesario (Italy)
Esposito, Mariana (Italy)
Formicola, Valerio (Italy)
Garofalo, Alessia (Italy)
Gentile, Ugo (Italy)
Getir, Sinem (Germany)
Hatebur, Denis (Germany)
Hutchison, Felix (USA)
Kane, Aaron (USA)

Karray, Hassen (Germany)
Laibinis, Linas (Finland)
Lechner, Jakob (Austria)
Lisagor, Oleg (UK)
Lollini, Paolo (Italy)
Lopatkin, Ilya (UK)
Machin, Mathilde (France)
Marques, Luis (Portugal)
Massini, Eugen (Germany)
Meis, Rene (Germany)
Menon, Catherine (UK)
Miler, Jakub (Poland)
Montecchi, Leonardo (Italy)
Naqvi, Syed Rameez (Austria)
Natella, Roberto (Italy)
Nostro, Nicola (Italy)
Núñez, David (UK)
Pai, Ganesh (USA)
Papanikolaou, Nick (UK)

Paragliola, Giovanni (Italy)

Pathan, Risat (Sweden)

Pecchia, Antonio (Italy)

Pereverzeva, Inna (Finland)

Pietrantuono, Roberto (Italy)

Polzer, Thomas (Austria)

Prokhorova, Yuliya (Finland)

Resch, Stefan (Austria)

Sicuranza, Mario (Italy)

Sobesto, Bertrand (USA)

Stroud, Robert (UK)

Tarasyuk, Anton (Finland)

Taylor, Malcolm (USA)

Veeravalli, Varadan Savulimedu
(Austria)

Zizyte, Milda (USA)

Zulianello, Marco (The Netherlands)

Sponsoring Institutions

European Workshop on Industrial Computer Systems
Reliability, Safety and Security



Laboratory for Analysis and Architecture of Systems,
Carnot Institute



Centre national de la recherche scientifique



International Federation for Information Processing



Université Paul Sabatier



University of Toulouse



Région Midi-Pyrénées



Communauté urbaine Toulouse Métropole



Mairie de Toulouse



Sciences et Technologies pour l'Aéronautique et
l'Espace



Airbus EADS



Institut National des Sciences Appliquées de Toulouse



European Research Consortium for Informatics and Mathematics



Austrian Institute of Technology



Société de l'Electricité, de l'Electronique et des Technologies
de l'Information et de la Communication



World competitiveness Cluster in Aeronautics, Space and Embedded Systems, of Midi-Pyrenées and Aquitaine



Informationstechnische Gesellschaft



German Computer Society



Austrian Computer Society



European Network of Clubs for Reliability and Safety of Software-Intensive Systems



Table of Contents

Invited Paper

Logic and Epistemology in Safety Cases	1
<i>John Rushby</i>	

Safety Requirements and Assurance

Comparative Conformance Cases for Monitoring Multiple Implementations of Critical Requirements	8
<i>Janusz Górski, Aleksander Jarzębowicz, and Jakub Miler</i>	
A Formal Basis for Safety Case Patterns	21
<i>Ewen Denney and Ganesh Pai</i>	

Testing and Verification

Testing Autonomous Robot Control Software Using Procedural Content Generation	33
<i>James Arnold and Rob Alexander</i>	
Fine-Grained Implementation of Fault Tolerance Mechanisms with AOP: To What Extent?	45
<i>Jimmy Lauret, Jean-Charles Fabre, and H�el�ene Waeselynck</i>	
Formalisation of an Industrial Approach to Monitoring Critical Data . . .	57
<i>Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Dubravka Ili�c, and Timo Latvala</i>	

Security

Protecting Vehicles Against Unauthorised Diagnostics Sessions Using Trusted Third Parties	70
<i>Pierre Kleberger and Tomas Olovsson</i>	
Vulnerability Analysis on Smart Cards Using Fault Tree	82
<i>Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet</i>	
Does Malware Detection Improve with Diverse AntiVirus Products? An Empirical Study	94
<i>Iliir Gashi, Bertrand Sobesto, Vladimir Stankovic, and Michel Cukier</i>	

Software Reliability Assessment

Software Fault-Freeness and Reliability Predictions	106
<i>Lorenzo Strigini and Andrey Povyakalo</i>	
Does Software Have to Be Ultra Reliable in Safety Critical Systems? . . .	118
<i>Peter Bishop</i>	

Practical Experience Reports and Tools I

The SafeCap Platform for Modelling Railway Safety and Capacity	130
<i>Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky</i>	
Embedded System Platform for Safety-Critical Road Traffic Signal Applications	138
<i>Thomas Novak and Christoph Stoegerer</i>	
Low-Level Attacks on Avionics Embedded Systems	146
<i>Anthony Dessiatnikoff, Eric Alata, Yves Deswarte, and Vincent Nicomette</i>	

Safety Assurance in Automotive

Safety Cases and Their Role in ISO 26262 Functional Safety Assessment	154
<i>John Birch, Roger Rivett, Ibrahim Habli, Ben Bradshaw, John Botham, Dave Higham, Peter Jesty, Helen Monkhouse, and Robert Palin</i>	
Structuring Safety Requirements in ISO 26262 Using Contract Theory	166
<i>Jonas Westman, Mattias Nyberg, and Martin Törngren</i>	

Error Control Codes

Flexible Unequal Error Control Codes with Selectable Error Detection and Correction Levels	178
<i>Luis-J. Saiz-Adalid, Pedro-J. Gil-Vicente, Juan-Carlos Ruiz-García, Daniel Gil-Tomás, J.-Carlos Baraza, and Joaquín Gracia-Morán</i>	
Safety Transformations: Sound and Complete?	190
<i>Ute Schiffel</i>	

Invited Paper

It Is (Almost) All about Human Safety: A Novel Paradigm for Robot Design, Control, and Planning	202
<i>Sami Haddadin, Sven Parusel, Rico Belder, and Alin Albu-Schäffer</i>	

Dependable User Interfaces

Understanding Functional Resonance through a Federation of Models: Preliminary Findings of an Avionics Case Study	216
<i>Célia Martinie, Philippe Palanque, Martina Ragosta, Mark Alexander Suján, David Navarre, and Alberto Pasquini</i>	

Model-Based Development of the Generic PCA Infusion Pump User Interface Prototype in PVS	228
<i>Paolo Masci, Anaheed Ayoub, Paul Curzon, Insup Lee, Oleg Sokolsky, and Harold Thimbleby</i>	

Practical Experience Reports and Tools II

Characterization of Failure Effects on AADL Models	241
<i>Bernhard Ern, Viet Yen Nguyen, and Thomas Noll</i>	

Derived Hazard Analysis Method for Critical Infrastructures	253
<i>Thomas Gruber, Georg Neubauer, Andreas Weinfurter, Petr Böhm, and Kurt Lamedschwandner</i>	

A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution	265
<i>Fatemeh Ayatollahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson</i>	

Hazard and Failure Mode Analysis

OpenMADS: An Open Source Tool for Modeling and Analysis of Distributed Systems	277
<i>Ermeson C. Andrade, Marcelo Alves, Rubens Matos, Bruno Silva, and Paulo Maciel</i>	

A Controlled Experiment on Component Fault Trees	285
<i>Jessica Jung, Andreas Jedditschka, Kai Höfig, Dominik Domis, and Martin Hiller</i>	

DFTCALC: A Tool for Efficient Fault Tree Analysis	293
<i>Florian Arnold, Axel Belinfante, Freark Van der Berg, Dennis Guck, and Mariëlle Stoelinga</i>	

Author Index	303
-------------------------------	------------

Logic and Epistemology in Safety Cases

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. A safety case must resolve concerns of two different kinds: how complete and accurate is our knowledge about aspects of the system (e.g., its requirements, environment, implementation, hazards) and how accurate is our reasoning about the design of the system, given our knowledge.

The first of these is a form of epistemology and requires human experience and insight, but the second can, in principle, be reduced to logic and then checked and automated using the technology of formal methods.

We propose that reducing epistemic doubt is the main challenge in safety cases, and discuss ways in which this might be achieved.

1 Introduction

Different industries take different approaches to software assurance and certification, but no matter what form a submission for certification actually takes, it can be understood and examined within the framework of a *safety case*. More specifically, we can suppose there are safety-relevant *claims* for the system concerned, some *evidence* about its assumptions, design, and construction, and an *argument*, based on the evidence, that the claims are satisfied. The standards and guidelines on which some certification processes are based often specify only the evidence to be produced; the claims and argument are implicit, but presumably informed the deliberations that produced the guidelines. There is current work to make explicit the safety cases implicit in some guidelines: for example, the FAA has sponsored work at NASA to do this for the airborne software guidelines DO-178C [1].

The safety case for any interesting system will be large, and one wonders how reliable the processes of review and evaluation—and ultimately of certification—can be for such large artifacts. The Nimrod case demonstrates that these concerns are not idle [2]. In this paper, I propose that modern formal methods and the techniques of automated deduction allow the argument of a safety case to be evaluated by systematic and largely automated methods. That is to say, evaluation of a safety case argument can—and should—largely be reduced to calculation, in a modern application of Leibniz’ vision:

“If controversies were to arise, there would be no more need of disputation between two philosophers than between two accountants. For it

would suffice to take their pencils in their hands, to sit down to their slates, and to say to each other... ‘Let us calculate’.”

This proposal is not intended to denigrate or displace human judgment and insight, but to liberate these from an essentially clerical task so that they may be focused on areas that really need them. In what follows, I hope to identify those areas where human judgment is best deployed, and how a formalized safety case can best be organized to facilitate this.

2 Logic Doubt

The argument of a safety case establishes claims, based on evidence, and this has obvious parallels with formal proof, which justifies a conclusion, based on premises. Modern methods of formal specification and verification provide notations adequate to the formalization of premises and conclusions concerning complex systems, and allow substantial automation to be applied to the construction of formal proofs. By now, formal verification has been applied to many designs, algorithms, and software for critical systems (e.g., [3–5]), so this invites the question: “what is the difference between a formalized safety case and a formal verification?” In my opinion, the answer has two parts.

First, formal verification is generally concerned with *correctness*, whereas a safety case is concerned with *perfection*. Correctness is the more limited notion; for software systems, it is usually a demonstration that the high-level software requirements are properly implemented by the corresponding programming language source code, which may be in C, SCADE, or Stateflow/Simulink, etc. Perfection, on the other hand, is an all-encompassing notion: it asks whether the high-level software requirements are themselves correct (relative to the more abstract system requirements), and whether the software as it runs (with the support of the relevant compilers, linkers, middleware, operating system, and hardware) really has the behavior assumed by the semantics of the programming languages concerned, and satisfies the safety objectives identified in the system requirements. A perfect system is one that will never suffer a (safety-relevant) failure. This use of the term “perfection” comes from literature on the nature of software assurance and the statistical basis for certification [6].

In many industries, assurance of software correctness is performed and assessed by different groups and according to different guidelines than assurance of perfection for the subsystem of which it is a part. In commercial aircraft, for example, the system-level requirements are developed and assessed (typically by systems engineers) relative to the guidelines known as ARP 4761 [7] and ARP 4754A [8], whereas software is developed and assessed (typically by software engineers) relative to the guidelines known as DO-178B [9], or its successor DO-178C [1]. One of the top-level “objectives” of DO-178B and DO-178C is to establish that the high-level software requirements are consistent with the system-level requirement; thereafter, DO-178B is about correctness, all the way down to the running program.

The second part of the answer to the difference between a formal verification and a safety case is that a formal verification generally takes the formalized premises and conclusion as given and is focused on the demonstration that the latter follows from the former. A safety case will add to this careful justification that the premises are true of the system concerned, and that the formal conclusion does indeed support the real-world interpretation required of it. An instructive illustration of these differences between a formal verification and a “case” is provided by formal verification of the Ontological Argument [10]: this is a formally correct proof of the existence of God—a claim that will cause most readers to examine its premises and the formalization of its conclusion with especial interest.

The two parts to this answer are different sides of the same coin: a safety case tackles a bigger problem (i.e., perfection) than verification (resp. correctness), so there is more to do. What is being proposed here may seem like enlarging the scope of formal verification from correctness to perfection. But, of course, those topics that traditionally are excluded from formal verification and left to the larger safety case are (mostly) excluded for good reasons: generally, they do not lend themselves to formal analysis, or they concern interpretation of the formal analysis itself. So what is proposed here is not the same as expanding verification from correctness to perfection but, instead, augmenting formal verification to include the skeleton of the rest of the safety argument.

By this I mean that we wish to use formal methods to keep track of safety case claims, and what follows from what, and why, but we do not expect to reduce those down to self-evident axioms and formalized theories. For example, to record why some formally verified theorem is considered to discharge an informally stated safety claim we may employ a proof rule equivalent to “because an expert says so,” with the expert’s informal justification attached as a comment. For another example, the top-level of the safety case might be organized as an enumeration over hazards; this organization could be justified by reference to a collection of patterns for safety-cases arguments, while the list of hazards is justified by another “because an expert says so” proof rule, with the process of hazard discovery attached as a comment. This approach to formalization of safety cases is developed in more detail in a previous paper [11]. The idea is that the full resources of automated deduction within a formal verification system become available for the purpose of evaluating the argument of a safety case. The description in [11] envisaged augmenting a formal verification system to support this activity (chiefly by providing ways to manage the narrative comments justifying informal proof steps), but a recent alternative is to use a framework such as the “Evidential Tool Bus” (ETB) [12], which is designed to assemble formal claims from multiple sources.

My suggestion is that an approach like this could largely eliminate “logic doubt” as a concern when evaluating a safety case. Evaluators would have not only the soundness guarantee of the relevant verification system or ETB, but could actively probe the argument using “what-if” exploration (e.g., temporarily remove or change an assumption and observe how the proof fails, or inspect

a counterexample). It is worth noting that proponents of safety cases sometimes recommend Toulmin’s approach to framing arguments [13] rather than traditional logic [14]. Toulmin stresses justification rather than inference and adds “qualifier,” “backing,” and “rebuttal” components to the traditional “warrant” (i.e., proof) in presenting the derivation of a claim from evidence. Toulmin was writing in the 1950s, when mechanized support for argument was barely more practicable than it was in Leibniz’ day and, in my opinion, his approach reflects the limitations of the technology at his disposal (basically, the printed page), where the arguer must attempt to anticipate and prepare responses to objections and challenges by those he would persuade but will not be able to interact with in real time. Nowadays, automated deduction does allow real-time interaction and I believe this provides a better vehicle for persuasion and exploration of logic doubt than Toulmin’s approach.

3 Epistemic Doubt

If we now contemplate the tasks facing an evaluator who must appraise a formalized safety case of the kind advocated above, we see that “logic doubt” is largely eliminated and concern will instead focus on the leaves of the argument. In a conventional formal verification, these would be the formal models and axioms comprising the premises of the verification, together with questions about their interpretation and that of the conclusion; in the expanded treatment that formalizes the safety case, the leaves will also include informal expert justification. I claim that all of this leaf-level material is *epistemic* in nature: that is to say, it concerns our knowledge about the system and its place in the world, including its context, requirements, environment, design, construction, hazards, and everything else that is germane to its safety.

The claim in the previous paragraph makes explicit an observation that I believe has long been accepted implicitly: there are just two kinds of concern underlying system safety, and other similar kinds of system analysis: those that are epistemic in nature (i.e., concerning our knowledge) and those that are about logic (i.e., our reasoning, based on that knowledge). As evidence, I cite the traditional partitioning of system assurance into verification and validation: the former (“did I build the system right?”) is about logic, while the latter (“did I build the right system?”) is about epistemology.

If mechanically supported formal reasoning allows us to reduce logic doubts, then the remaining opportunity for improving safety cases is to reduce epistemic doubt. This means that large informal justifications attached as comments to proof rules of the flavor “because an expert says so” should be broken into more manageable pieces connected by explicit reasoning. Furthermore, we should strive to find ways to represent expert knowledge in ways that support examination and validation while, at the same time, it is directly useful in the formalized argument of the case. In essence, this means we should represent our knowledge in logic. Software *is* logic, so there is, in principle, no obstacle to representing its epistemology (requirements, specification, code, semantics) in logic: that is why formal verification is feasible—and increasingly practical and cost-effective—for software.

The world with which the software interacts—the world of devices, machines, people and institutions—is not (outside of analytic philosophy) typically considered a manifestation of applied logic, but I believe there are indications that it can be. It is increasingly common that system developers build models of the world using simulation environments such as Simulink/Stateflow. These models represent their epistemology, which they refine and validate by conducting simulation experiments. Other simulation environments, developed for human-computer-interaction (HCI) studies, model aspects of human behavior as well as machines [15, 16], and models of larger human organizations are employed in many fields ranging from disaster planning to economics and politics.

The simulation environments that support these modeling activities are themselves software and they could, with difficulty, be represented within a logic-based assurance framework (this is already feasible for Simulink, whose models can be imported into many verification environments [17]). However, this is not the main obstacle to the use of simulation models within formalized assurance cases. Rather, the problem is that simulation models are designed for that purpose and simultaneously say too much and too little for the purposes of assurance and minimization of epistemic doubt. For example, the Simulink model for a car braking system will provide equations that allow calculation of the exact rate of deceleration in given circumstances (which is more information than we need), but will not provide (other than indirectly) the maximum stopping distance—which is an example of a property that may be needed in an assurance case. The crucial point is that it should be easier to resolve epistemic doubts about a simple constraint, such as maximum stopping distance, than the detailed equations that underlie a full simulation model.

So my proposal is that for the purpose of recording the epistemology of a safety case, models should be expressed as systems of constraints rather than as simulation models: less is more. Until fairly recently, it would have been difficult to validate systems of constraints: unlike simulation models, it was not feasible to run experimental calculations to check the predictions of the model against intuition and reality. Fortunately, we now have technology such as “infinite bounded model checkers,” based on highly effective constraint solvers for “satisfiability modulo theories” (SMT) that allow exploration of constraint-based models (see [18, 19] for some simple examples).

I believe that constraint-based models of this kind could be particularly useful in validating system-level requirements. All software-related incidents in commercial aircraft have their origin in imperfect system-level requirements, or in mismatches between the system-level requirements and top-level software requirements [20], both of which may be due, in part, to lack of good notations and tools for representing and validating system knowledge at this level of abstraction.

4 Research Directions

The diagnosis and proposals in this paper are deliberately speculative and, perhaps, provocative. The basic claim is that evaluation of large safety cases will

benefit from—indeed, requires—automated assistance: as Leibniz said, rather than contemplation and discussion, “let us calculate.” I argued that there are just two components, and hence two sources of doubt, in a safety case: our epistemology (what we know about the system, its context, and its environment) and our logic (the validity of our reasoning about the safety of the system, given our knowledge). Formal verification systems provide tools that can be adapted to represent, analyze, and explore the logic of our case, thereby largely eliminating logic doubt, so that the tougher challenge is to represent our epistemology in a form that can be used by such systems. This can be accomplished by building models in logic that describe the elements of our knowledge (the behavior of the environment etc.); these models should be described as systems of constraints (rather than, say, simulation models) and these can be explored and validated using modern tools based on SMT solvers.

Suggested research directions are simple: try this out and see if it works. I suspect that it will be much more difficult to find ways to justify adequate completeness of our epistemology (e.g., that we have identified *all* hazards) than its validity, and that this will pose a challenging research problem.

Acknowledgements. This work was supported by NASA under contract NNA13AB02C with Drexel University and by the DARPA HACMS program under contract FA8750-12-C-0284 with AFRL. The content is solely the responsibility of the author and does not necessarily represent the official views of NASA or DARPA.

References

1. Requirements and Technical Concepts for Aviation (RTCA) Washington, DC: DO-178C: Software Considerations in Airborne Systems and Equipment Certification (2011)
2. Haddon-Cave, C.: The Nimrod Review: An independent review into the broader issues surrounding the loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006. Report, The Stationery Office, London, UK (2009), <http://www.official-documents.gov.uk/document/hc0809/hc10/1025/1025.pdf>
3. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220. ACM (2009)
4. Miner, P., Geser, A., Pike, L., Maddalon, J.: A unified fault-tolerance protocol. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT 2004. LNCS, vol. 3253, pp. 167–182. Springer, Heidelberg (2004)
5. Narkawicz, A., Muñoz, C.: Formal verification of conflict detection algorithms for arbitrary trajectories. *Reliable Computing* 17, 209–237 (2012)
6. Littlewood, B., Rushby, J.: Reasoning about the reliability of diverse two-channel systems in which one channel is “possibly perfect”. *IEEE Transactions on Software Engineering* 38, 1178–1194 (2012)

7. Society of Automotive Engineers: Aerospace Recommended Practice (ARP) 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (1996)
8. Society of Automotive Engineers: Aerospace Recommended Practice (ARP) 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems (1996), Also issued as EUROCAE ED-79; revised as ARP 4754A (December 2010)
9. Requirements and Technical Concepts for Aviation (RTCA) Washington, DC: DO-178B: Software Considerations in Airborne Systems and Equipment Certification (1992), This document is known as EUROCAE ED-12B in Europe
10. Rushby, J.: The Ontological Argument in PVS. In: Shilov, N. (ed.) *Fun With Formal Methods*, St Petersburg, Russia (2013), Workshop in association with CAV 2013
11. Rushby, J.: Formalism in safety cases. In: Dale, C., Anderson, T. (eds.) *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*, Bristol, UK, pp. 3–17. Springer (2010)
12. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool Integration with the Evidential Tool Bus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) *VMCAI 2013*. LNCS, vol. 7737, pp. 275–294. Springer, Heidelberg (2013)
13. Toulmin, S.E.: *The Uses of Argument*, Updated edition. Cambridge University Press (2003) (the original is dated 1958)
14. Bishop, P., Bloomfield, R., Guerra, S.: The future of goal-based assurance cases. In: *DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, Florence, Italy (2004)
15. Pritchett, A.R., Feigh, K.M., Kim, S.Y., Kannan, S.: Work Models that Compute to support the design of multi-agent socio-technical systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans* (under review)
16. Bolton, M.L., Bass, E.J.: Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In: *IEEE International Conference on Systems, Man, and Cybernetics*, Anchorage, AK, pp. 1788–1794 (2011)
17. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. *Communications of the ACM* 53, 58–64 (2010)
18. Rushby, J.: A safety-case approach for certifying adaptive systems. In: *AIAA Infotech@Aerospace Conference*, Seattle, WA, American Institute of Aeronautics and Astronautics (2009) AIAA paper 2009-1992
19. Bass, E.J., Feigh, K.M., Gunter, E., Rushby, J.: Formal modeling and analysis for interactive hybrid systems. In: *Fourth International Workshop on Formal Methods for Interactive Systems: FMIS 2011*, Limerick, Ireland. *Electronic Communications of the EASST*, vol. 45 (2011)
20. Rushby, J.: New challenges in certification for aircraft software. In: Baruah, S., Fischmeister, S. (eds.) *Proceedings of the Ninth ACM International Conference on Embedded Software: EMSOFT*, Taipei, Taiwan, pp. 211–218. Association for Computing Machinery (2011)

Comparative Conformance Cases for Monitoring Multiple Implementations of Critical Requirements

Janusz Górski^{1,2}, Aleksander Jarzębowicz^{1,2}, and Jakub Miler^{1,2}

¹ Department of Software Engineering, Gdansk University of Technology, Poland

² NOR-STA Project, Gdansk University of Technology, Poland

{jango,olek,jakubm}@eti.pg.gda.pl

www.nor-sta.eu

Abstract. The paper presents the concept and the mechanism of comparative conformance cases which support conformance monitoring in situations where a standard or other set of requirements are being implemented at multiple sites. The mechanism is enabled by NOR-STA services which implement the TRUST-IT methodology and are deployed in the cloud in accordance with the SaaS model. In the paper we introduce the concept of comparative conformance cases, explain the software services used to implement them and present a case study of monitoring the implementation of the EC Regulation No. 994/2010, related to risk management of gas supply infrastructures across Europe.

Keywords: conformance case, conformance monitoring, critical infrastructures protection, trust cases, NOR-STA services.

1 Introduction

Regulations and standards are among the important mechanisms through which the European program of risk governance for the ICT and energy sectors is being implemented [1]. To make these mechanisms effective, it is important not only to promote implementation of standards and regulations but also to assess the actually achieved level of conformance and to continuously monitor the conformance across the different critical infrastructure stakeholders.

In this paper we introduce *comparative conformance case* – a mechanism that supports conformance monitoring in situations where a standard, directive or other set of requirements are implemented at multiple sites. Hereafter, we will call the source of conformance requirements a ‘standard’. If used by the party in charge of supervising implementation of a standard, the mechanism provides means to review the evidence supporting conformance and to review and assess the related conformance cases against a selected assessment scale.

Comparative conformance case is based on the concept of a *trust case* [2, 3] which extends the concept of *safety case* [4] commonly used in the safety-critical domain to justify safety properties of various systems, for instance avionic, nuclear, automotive, medical, military and so on [5, 6]. The concept of safety case has been extended

(under the name *assurance case*) to cover any critical property to be assured, like safety, security, reliability and others [7]. Under the name of *trust case* it has been further generalized and refers to the situations where the focus is on a selected feature to be demonstrated, not necessarily being ‘critical’.

We are particularly interested in situations where a common argumentation structure is shared by numerous concrete cases. As an example take a standard and the question of conformance demonstration. Then, the structure of the conformance case may be common for multiple implementations of the standard and the difference between particular implementations is mostly in the evidence supporting the argument. This observation led to the concept of the *conformance case template* which can have multiple instantiations where each instance, after attaching the relevant evidence, becomes a complete *conformance case* [8, 9]. The concept of comparative conformance case builds upon these notions.

In the paper we outline the TRUST-IT methodology of developing and assessing trust cases and the related NOR-STA platform which supports this methodology by offering a set of software services in the computing cloud. Next, we introduce the mechanism of comparative conformance cases and the related scenario of its application. Then, we present a case study demonstrating implementation of this scenario with the objective of supporting monitoring the conformance to the European Commission Regulation No. 994/2010 related to risk management of gas supply infrastructures in different EU Member States.

2 TRUST-IT Methodology and NOR-STA Services

TRUST-IT [2, 3, 8] is an approach to promoting trust by developing, maintaining and presenting on-line arguments demonstrating trustworthiness. An argument can be published, edited and assessed, and it is visualized in a graphical form together with the result of the assessment of its ‘compelling power’. Evidence integrated with an argument is kept in digital documents of any form: text, graphics, image, web page, video, audio and so on. The evidence supports what an argument postulates about the state of the world. In TRUST-IT terminology, such postulates are called ‘facts’. Depending on the support given by the evidence to the corresponding facts, the argument is more or less convincing. TRUST-IT introduces a model of an argument (following [10]), a graphical language for expressing arguments, and a technique for integrating arguments with evidence (see Fig. 1). The arrows linking the nodes shown in Fig. 1 represent the can-be-child-of relationship in the argument tree. The abstract argument model of TRUST-IT is similar to GSN [11] and CAE [12], the differences are more on a technical and representation levels.

Argument conclusion is represented by a *claim* node. A node of type *argumentation strategy* links the *claim* with the corresponding premises and uses a *rationale* to explain and justify the inference leading from the premises to the claim. A premise is a sort of assertion and can be of the following type: an *assumption* represented by an assertion assumed to be true which is not further justified; a *claim* represented by an assertion to be further justified by its own premises; and a *fact* represented by an

assertion to be demonstrated by the supporting evidence. The evidence is integrated by nodes of type *reference* which point to external resources (files of any type, web pages, etc.). In addition, *information* nodes (denoted **i**) can be used in any place to provide explanatory information. This model can generate trees of arbitrary depth where the root of the tree is the top-most claim and the leaves are references pointing to the evidence supporting facts, assumptions and/or rationales of selected argumentation strategies (in our experience we are dealing with arguments of up to several thousand nodes).

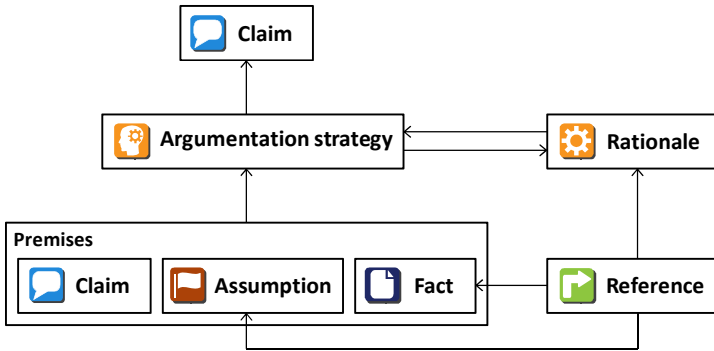


Fig. 1. The TRUST-IT model of argument

By analyzing the inferences and checking the evidence supporting facts, an auditor can work out his/her opinion about how strong the argument is towards the conclusion, and where its weaknesses and strengths are. TRUST-IT supports this activity in different ways. The most advanced is the argument appraisal mechanism based on Dempster-Shafer theory of evidence [13] and the corresponding mechanism of visualization of the argument compelling power [14]. Here, the auditor can express his/her appraisals referring to so called *opinion triangle* shown in Fig. 2.

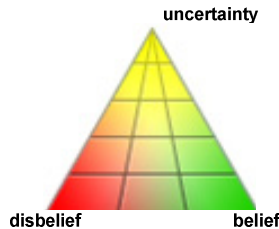


Fig. 2. The opinion triangle for issuing argument appraisals based on Dempster-Shafer theory

By choosing a position within the triangle, the auditor, after examining the evidence supporting a fact, decides to which extent he/she accepts/rejects the fact and what is the level of uncertainty associated with this opinion. Similar appraisals can be issued with respect to the inferences used in the argument, represented by the related

nodes of *rationale* type. The aggregation rules (see [14] for the details) provide for automatic appraisal of all claims of the argument (including the top-most one), provided the appraisals of all facts, assumptions and rationales have already been issued.

In addition to the above, TRUST-IT provides for other, user-defined argument appraisal mechanisms (involving different assessment scales and aggregation rules). These mechanisms can be activated according to the users' needs. For instance, in one of our case studies of arguments related to standards conformance the stakeholders decided that a simple three-state scale {non-compliant, partially-compliant, compliant} was in use.

TRUST-IT arguments have already been (among others) developed to: analyze safety, privacy and security of personalized health and lifestyle oriented services [15], monitor the environmental risks [16] and support standards conformance for health, business and administration sectors [17].

Application of TRUST-IT is supported by the NOR-STA platform of software services. The services are deployed in accordance with the SaaS (Software as a Service) cloud computing model. The scope of functionalities of NOR-STA services includes: argument representation and editing using the graphical symbols shown in Fig. 1, integration (through references) of various types of evidence, argument assessment and visualization of the assessment results, publishing of an argument, and evidence hosting in protected repositories.

3 Comparative Conformance Case

TRUST-IT approach is generic and can be applied in any context where evidence based argumentation brings added value to decision making processes and disputes. One such application area is standards conformance where a standard's user is expected to construct and present an argument demonstrating conformance. While applied to standards conformance, TRUST-IT introduces additional, more specific concepts [8, 9]. *Conformance case template* is an argumentation structure derived from a standard. This structure is common for all conformance cases related to the standard. It explicitly identifies placeholders for the supporting evidence and may indicate places where more specific, implementation dependent argumentation is to be provided. Template development involves domain experts representing the standard's owner and standard's auditor viewpoints. *Conformance case* is a complete argument which is developed from the template providing the required evidence and possibly by appending a more specific argumentation. *Conformance assessment* is an act of assigning appraisals to the conformance argument components to assess their 'compelling power'.

The relationship between the conformance case template and a conformance case following the structure of the template is shown in Fig.3. Filled rectangles denote nodes already included in the template, the hollow rectangles denote nodes added as more specific argumentation and the ovals represent the evidence supporting the conformance case. The concept of case template is similar to argument schemes/patterns

[5, 6] used in safety cases (a template suggests how to demonstrate conformance to entire set of standard's requirements whereas a pattern suggests how to demonstrate a single, specific type of claim). However, the conformance case template is far richer than just providing an argumentation scheme. In addition it includes the source documentation of the related standard, examples of good practices in structuring the evidence, single evidence placeholders referenced in multiple requirements, explicit interdependencies between standard fragments and many others.

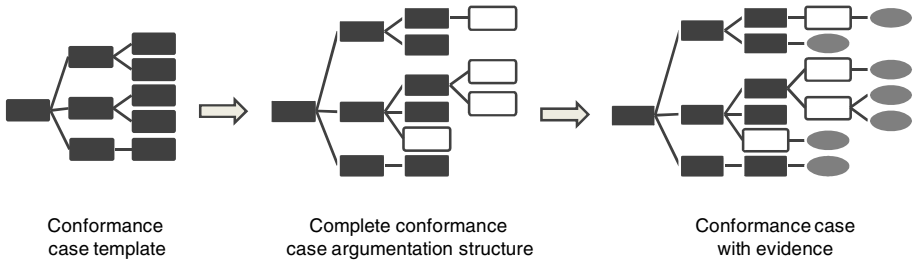


Fig. 3. The conformance case template – conformance case relationship

Let us assume that a given conformance case template is being used by a number of users, and each of them has developed her/his own conformance case based on the template. Then, each of the cases can be reviewed in the node-by-node manner, for instance, by accessing the facts and verifying the evidence supporting each fact.

Now, let us assume that in addition to the users developing their own conformance cases, there is a separate body, call it *supervisor*, who is in charge of monitoring all the cases and possibly assessing how strongly the claims and facts listed in the conformance case template are supported in different cases.

The concept of *comparative conformance case* embodies the idea that having a conformance case template in an explicit form, one can point to a selected node of the template and in response will have access to the structure demonstrating how this node is represented in each conformance case derived from the template. For instance, by pointing to a given fact included in the template, the supervisor will be able to review, compare and assess the evidence submitted to demonstrate this fact in different conformance cases. And by pointing to a claim the supervisor will see the assessment results of this claim, for different conformance cases.

Implementing the concept of comparative conformance case would result in an interface with the following functionality:

- selecting the conformance cases to be compared;
- selecting a fact (claim) of the conformance case template which results in obtaining access to the evidence supporting this node in different conformance cases and to the appraisals of how strong this support is;
- accessing and browsing the evidence;
- issuing/changing appraisals of the support given by the evidence.

The above idea is illustrated in Fig. 4. The arrows shown in the picture point to a selected node of the conformance case template and to the corresponding nodes of the supervised conformance cases.

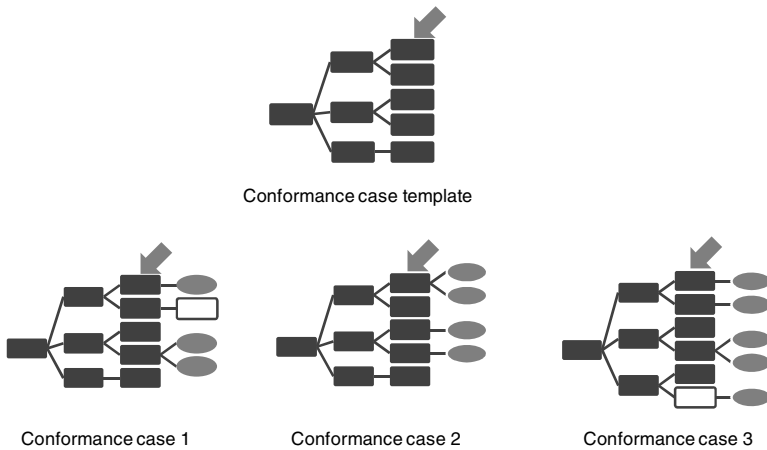


Fig. 4. Illustration of the comparative case concept

The concept of comparative conformance case brings added value in situations where monitoring the implementation of multiple conformance cases is of particular business relevance. In such situation, the supervising body can simply activate a comparative conformance case and by choosing different claims and facts can get immediate insight into the evidence supporting the chosen criterion across selected cases. In some cases it can also facilitate appraisals, especially if they are issued in relative terms.

The present scope of functionality of NOR-STA services covers: conformance case template management, conformance case management, management of evidence repositories, argument appraisal and appraisal comparison management.

4 Case Study: Monitoring Implementation of EC Regulation 994

The functionality of comparative conformance cases has been built into NOR-STA services to provide support for the monitoring scenario outlined in the previous section. In this section we present this scenario applied to monitoring the implementation of the European Commission Regulation No. 994/2010 [18].

4.1 The Regulation

The Regulation 994/2010 refers to risk management of gas infrastructures in the EU Member States. Below are some citations from this document.

Natural gas is an essential component in the energy supply of the European Union, constituting one quarter of primary energy supply and contributing mainly to electricity generation, heating, feedstock for industry and fuel for transportation. [...] Given the importance of gas in the energy mix of the Union, the Regulation aims at demonstrating to gas customers that all the necessary measures are being taken to ensure their continuous supply, particularly in case of difficult climatic conditions and in the event of disruption. [...]

This Regulation establishes provisions aimed at safeguarding the security of gas supply by ensuring the proper and continuous functioning of the internal market in natural gas, by allowing for exceptional measures to be implemented when the market can no longer deliver the required gas supplies and by providing for a clear definition and attribution of responsibilities among natural gas undertakings, the Member States and the Union regarding both preventive action and the reaction to concrete disruptions of supply. This Regulation also provides transparent mechanisms, in a spirit of solidarity, for the coordination of planning for, and response to, an emergency at Member State, regional and Union levels.

The Regulation imposes several obligations on Member States as well as on EU administration. The obligations for Member States include: conducting a thorough Risk Assessment the results of which should be summarized in an adequate report, and establishing Preventive Action Plan and Emergency Plan to be presented to the European Commission. The responsibility of the Commission is to instantiate an effective mechanism of monitoring how the regulation is being implemented.

4.2 Comparative Conformance Case for Regulation 994/2010

The scenario of implementing the comparative conformance case for Regulation No. 994/2010 (hereafter called Regulation) involves the following steps (the Users are in Member States and the Supervisor acts on behalf of the EU Commission):

- Step 1** - development of the conformance case template by the Supervisor;
- Step 2** - submitting the conformance case template to the Users;
- Step 3** - development of conformance cases by the Users;
- Step 4** - monitoring of conformance cases by the Supervisor.

Below we illustrate how this scenario is implemented with NOR-STA services. To demonstrate the implementation of Step 1 we have developed the conformance case template deriving it from the text of the Regulation. Fig. 5 presents an overview of the template (the hierarchy of nodes develops from the left to the right) In Fig. 5, the fact labeled F1.1.4 is linked to two references labeled December 2011: Information about intergovernmental agreements and December 2011: Risk Assessment Report. These references point to the places where two different pieces of evidence are to be integrated, demonstrating that the party implementing the Regulation has already prepared a risk assessment report and that the necessary intergovernmental agreements are in place to reduce risk related to gas supply.

Implementation of Step 2 is demonstrated by creating, for each user of Regulation, a separate space where it can develop its own conformance case. In NOR-STA terminology, such space is called 'project'. Each such project is initially filled with the

conformance case template. In the following text we assume that three such projects have been created for three different conformance cases of ‘dummy’ countries A, B and C.

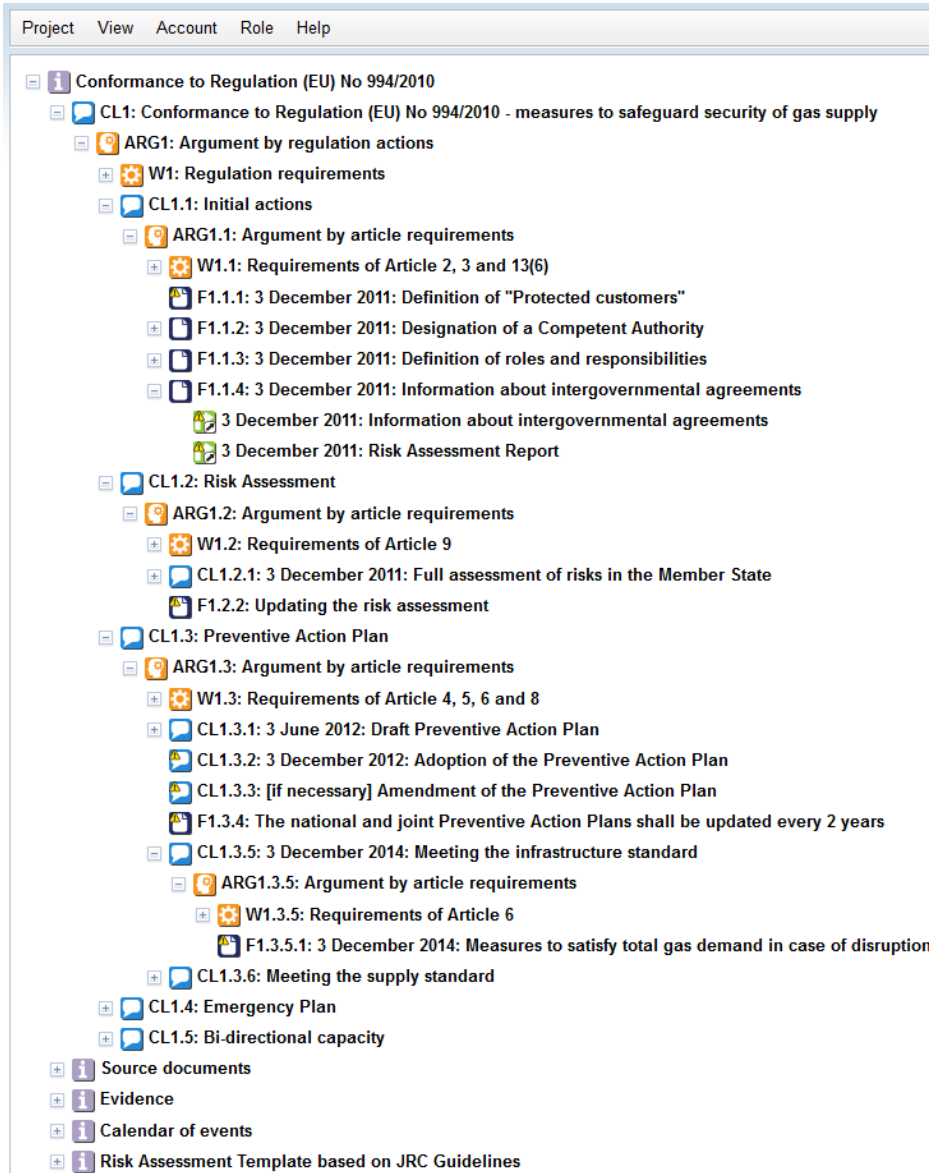


Fig. 5. Top-level decomposition of the Regulation conformance case template

Implementation of Step 3 is demonstrated by developing a separate conformance case, for each country. In Fig. 6 we can see the conformance case of Country A,

where some pieces of evidence has been already integrated. In the figure, the reference December 2011: Risk Assessment Report has been selected which resulted in opening the window with the referred evidence – the document of the risk assessment report. The remaining users (Country B and Country C) could have integrated their specific evidence in their own conformance cases in a similar way.

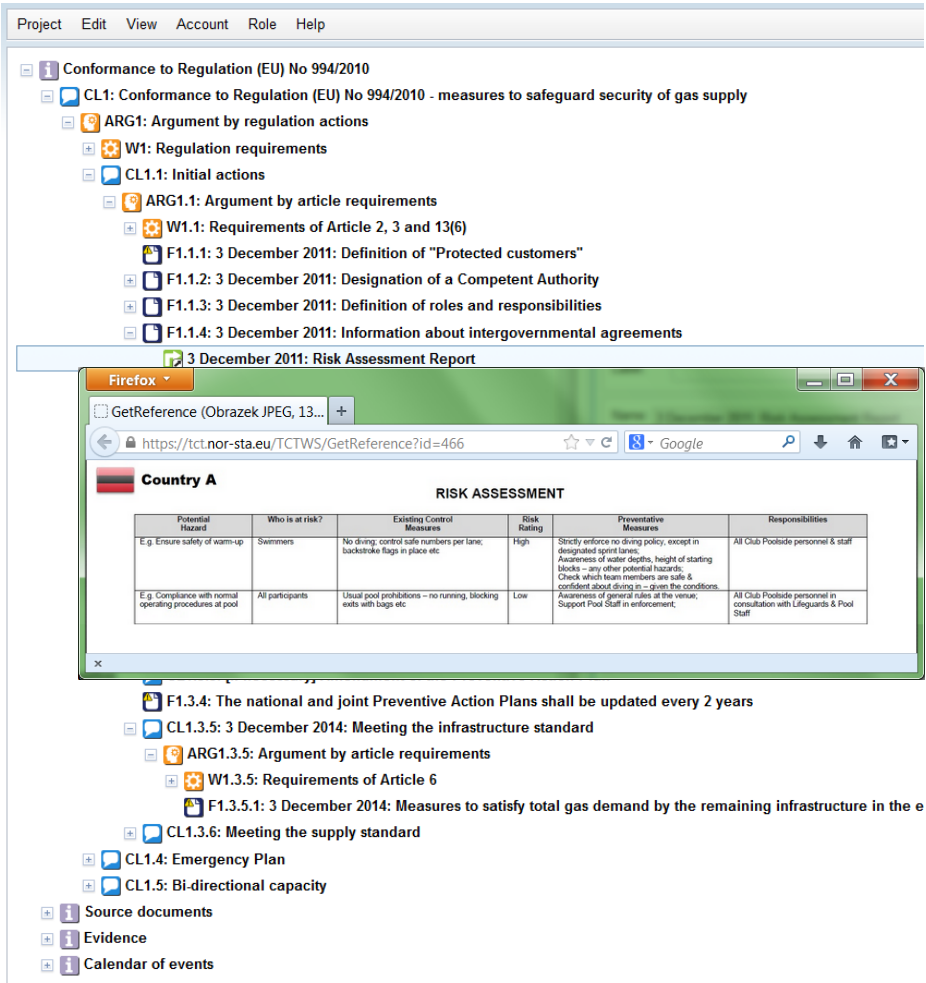


Fig. 6. An example evidence linked to fact F1.1.4

Implementation of Step 4 is demonstrated by opening the comparative panel for the fact F1.1.4. This is illustrated in Fig. 7. The panel gives the name of the related fact (region 1) and below (region 2) there are the evidence tiles, one for each country. The tiles indicate the format of the related evidence (an image for Country A, a .doc document for Country B and .pdf document for Country C). In addition, for each country,

the assessment of how well the evidence supports the analyzed fact is shown in the form of a colored circle. The colors indicate: acceptance (green), rejection (red) and uncertainty (yellow). Region 3 shows the opinion triangle related to the assessment of the evidence provided by Country A. The actual assessment is represented by the hollow circle in the triangle and the linguistic values corresponding to this assessment are presented beside the triangle (it reads: ‘with very high confidence the support given by the evidence is tolerable’). The evidence is shown in region 4.

The screenshot displays a software interface for comparing conformance cases. It features a menu bar (Project, View, Account, Role, Help) and a title bar (Assessment Comparison). The main content area is divided into four numbered regions:

- Region 1:** Project title and fact description: "FACT - F1.1.4: 3 December 2011: Information about intergovernmental agreements".
- Region 2:** Comparison of Country A, Country B, and Country C. It includes icons for Microsoft Word and Adobe PDF, and a small circular assessment indicator.
- Region 3:** Assessment panel showing "Belief", "Disbelief", and "Uncertainty" sliders. The "Confidence" slider is set to "with very high confidence", and the "Decision" slider is set to "tolerable". A "Delete Assessment" button is located at the bottom right.
- Region 4:** Two "RISK ASSESSMENT" tables. The top table is for Country A and the bottom table is for Country B. Both tables have columns for Potential Hazard, Who is at risk?, Existing Control Measures, Risk Rating, Preventative Measures, and Responsibilities.

Fig. 7. Comparative panel for fact F1.1.4

The assessments of facts can be issued either from the comparative panel, or otherwise by opening a selected conformance case and browsing its structure. Fig. 8 presents the conformance case of Country A together with the assessment of the facts being the premises of claim CL1.1. The bottom part of the picture gives the details of the assessment of fact F1.1.4 which is currently pointed to.

5 Conclusion and Future Work

The case study presented in this paper was the starting point for common experiments conducted in cooperation with the Institute for Energy and Transport of Joint

Research Centre (IET JRC) in Petten, the Netherlands, which is in charge of monitoring implementation of the Regulation 994/2010 in Member States, acting on behalf of the European Commission. The final conformance case template and the conformance cases for actual Member States are under development.

Gas distribution and supply is one of the key critical infrastructures requiring particular attention. With this example we demonstrated how the comparative conformance cases could be used to monitor implementation of the Regulation. For a selected normative document, the initial step is to create a conformance case template for this document which becomes a sort of ‘window’ through which the supervising body can look at different implementations of the norm to assess and compare the submitted evidence. The investment needed to create the template is moderate: in case of Regulation 994/2010 the initial template consisted of some 212 nodes and the total effort in its creation consumed 18 person-hours.

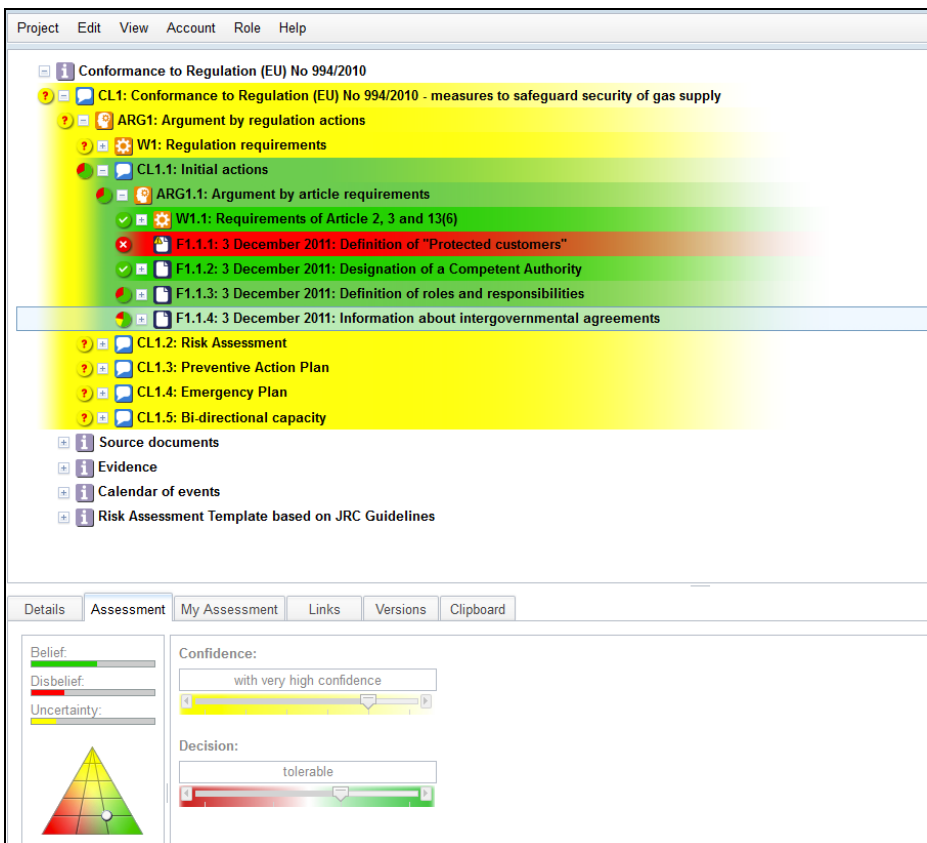


Fig. 8. Example assessment of the facts related to claim CL1.1

Conceptually, the user of a conformance case template can extend it by adding more specific argumentation (see Fig. 3). However, our experience with several standards (including healthcare related standards [17], standards for secure outsourcing, Common Assessment Framework [19] and others) shows that this option is rarely used. In majority of cases the template is being converted into a conformance case simply by supplying the evidence supporting particular facts.

Presently we are researching a possibility to use the comparative case concept to support monitoring of multiple cases related to the EU proposed legislation that would require oil and natural gas companies to submit emergency response plans and potential hazard reports before being given a license to drill offshore.

Acknowledgments. This work was partially supported by the NOR-STA project (grant no. UDA POIG.01.03.01-22-142/09-03). Contribution of M. Witkowicz, J. Czyżnikiewicz and P. Jar to technical implementation of the NOR-STA services and cooperation of M. Masera from JRC, Institute for Energy and Transport in establishing experiments related to Regulation 994/2010 are to be acknowledged.

References

1. Study on Risk Governance of European Critical Infrastructures in the ICT and Energy Sector, Final Report, AEA Technology, ED05761 (2009)
2. Górski, J.: Trust Case – a case for trustworthiness of IT infrastructures. In: *Cyberspace Security and Defense*. NATO Science Series, vol. 196, pp. 125–142. Springer (2005)
3. Górski, J., Jarzębowicz, A., Leszczyna, R., Miler, J., Olszewski, M.: Trust case: justifying trust in IT solution. *Reliability Engineering and System Safety* 89(1), 33–47 (2005)
4. Ministry of Defence, Defence Standard 00-56 Issue 4: Safety Management Requirements for Defence Systems (2007)
5. Yuan, T., Kelly, T.: Argument based approach to computer safety system engineering. *Int. J. Critical Computer-Based Systems* 3(3), 151–167 (2012)
6. Palin, R., Habli, I.: Assurance of automotive safety – A safety case approach. In: Schoitsch, E. (ed.) *SAFECOMP 2010*. LNCS, vol. 6351, pp. 82–96. Springer, Heidelberg (2010)
7. ISO/IEC 15026-2:2011: Systems and software engineering - Systems and software assurance - Part 2: Assurance case (2011)
8. Górski, J.: Trust-IT – a framework for trust cases, Workshop on Assurance Cases for Security - The Metrics Challenge. In: *Proc. of DSN 2007*, Edinburgh, UK, pp. 204–209 (2007)
9. Cyra, Ł., Górski, J.: Supporting Compliance with Safety Standards by Trust Case Templates. In: *Proc. ESREL 2007*, Stavanger, Norway, pp. 1367–1374 (2007)
10. Toulmin, S.: *The Uses of Argument*. Cambridge University Press (1958)
11. Goal Structuring Notation community Standard version 1 (2011)
12. Adelard Safety Case Editor (ASCE) website, <http://www.adelard.com/asce/>
13. Shafer G.: *Mathematical Theory of Evidence*. Princetown University Press (1976)
14. Cyra, Ł., Górski, J.: Support for argument structures review and assessment. *Reliability Engineering and System Safety* 96, 26–37 (2011)

15. Górski, J., Jarzębowicz, A., Miler, J., Witkiewicz, M., Czyżnikiewicz, J., Jar, P.: Supporting Assurance by Evidence-Based Argument Services. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP Workshops 2012. LNCS, vol. 7613, pp. 417–426. Springer, Heidelberg (2012)
16. Proceedings of the Workshop on Selected Problems in Environmental Risk Management and Emerging Threats, Gdansk, Poland (June 2009), <http://kio.pg.gda.pl/ERM2009/>
17. Górski, J., Jarzębowicz, A., Miler, J.: Validation of services supporting healthcare standards conformance. *Metrology and Measurements Systems XIX(2)*, 269–282 (2012)
18. Regulation (EU) No 994/2010 of the European Parliament and of the Council of 20 October 2010 concerning measures to safeguard security of gas supply and repealing Council Directive 2004/67/EC (2010)
19. European Institute of Public Administration, CAF-Common Assessment Framework (2012), <http://www.eipa.eu/en/topic/show/&tid=191>

A Formal Basis for Safety Case Patterns

Ewen Denney and Ganesh Pai

SGT / NASA Ames Research Center
Moffett Field, CA 94035, USA
{ewen.denney, ganesh.pai}@nasa.gov

Abstract. By capturing common structures of successful arguments, safety case patterns provide an approach for reusing strategies for reasoning about safety. In the current state of the practice, patterns exist as descriptive specifications with informal semantics, which not only offer little opportunity for more sophisticated usage such as automated instantiation, composition and manipulation, but also impede standardization efforts and tool interoperability. To address these concerns, this paper gives (i) a formal definition for safety case patterns, clarifying both restrictions on the usage of multiplicity and well-founded recursion in structural abstraction, (ii) formal semantics to patterns, and (iii) a generic data model and algorithm for pattern instantiation. We illustrate our contributions by application to a new pattern, the requirements breakdown pattern, which builds upon our previous work.

Keywords: Safety cases, Safety case patterns, Formal methods, Automation.

1 Introduction

Safety case patterns are intended to capture repeatedly used structures of successful (i.e., correct, comprehensive and convincing) arguments, within a safety case [11]. In effect, they provide a re-usable approach to safety argumentation by serving as a means to capture expertise, so-called *tricks of the trade*, i.e., known best practices, successful certification approaches, and solutions that have evolved over time. The existing notion of a pattern¹ is an extended argument structure, often specified graphically using the Goal Structuring Notation (GSN) [8], which abstractly captures the reasoning linking certain (types of) claims to the available (types of) evidence, and is accompanied by a clear prescription and proscription of its usage.

In current practice, patterns have informal semantics and, in general, they are given as descriptive non-executable specifications. Specifically, in existing tools, pattern-based reuse does not go beyond simple replication of a pattern argument structure, and manual replacement of its abstract elements with their concrete instances. Such usage is not only effort intensive but also unlikely to scale well. Algorithmically instantiating patterns is a natural solution to address this deficiency. However, to our knowledge, existing tools provide little to no such functionality, in part, because of the lack of a formal basis. The latter additionally impedes standardization and tool interoperability.

¹ In the rest of the paper we will simply use “pattern” instead of “safety case pattern”.

This paper extends the state of the art in safety case research through the following contributions: we give a formalization for argument structures elaborating on the nuances and ambiguities that arise when using the available GSN abstractions [8] for pattern specification. In particular, we clarify restrictions on the usage of multiplicity and extend the basic concepts to include a notion of well-founded recursion. Next, we give a formal semantics to patterns in terms of their (set of) concrete instantiations. We specify a generic data model and pattern instantiation algorithm, and illustrate their application to a new pattern: the requirements breakdown pattern, which builds upon our previous work [3], [6]. Specifically, both generalize and replace their previous incarnations [3] that mainly operated on requirements and hazard tables.

2 Background

Currently [10], [11], a pattern specification mainly contains:

- *Name*: the identifying label of the pattern giving the key principle of its argument.
- *Intent*: that which the pattern is trying to achieve.
- *Motivation*: the reasons that gave rise to the pattern.
- *Structure*: the abstract structure of the argument given graphically in GSN.
- *Participants*: each element in the pattern and its description.
- *Collaborations*: how the interactions of the pattern elements achieve the desired effect of the pattern.
- *Applicability*: the circumstances under which the pattern could be applied, i.e., the necessary context.
- *Consequences*: that which remains to be completed after pattern application.
- *Implementation*: how the pattern should be applied.

In addition, previously known usages, examples of pattern application, and related patterns are also given to assist in properly deploying a particular pattern. A variety of such pattern specifications can be found in [1], [10], and [15].

We assume that the reader is familiar with the basic syntax of GSN and do not repeat it here. The GSN standard [8] provides two types of abstractions for pattern specification: *structural* and *entity*.

Structural abstraction, which applies to the *is-solved-by* and the *in-context-of* GSN relations, is supported by the concepts of *multiplicity* and *optionality*. The former generalizes n -ary relations between GSN elements, while the latter captures alternatives in the relations, to represent a k -of- m choice, where $k \geq 1$. There are, further, two types of multiplicity: *optional*, implying zero or one, and *many*, implying zero or more. Multiplicity can be combined with optionality: placing a multiplicity symbol prior to the option describes a multiplicity over all the options. This is equivalent to placing that multiplicity symbol on all the alternatives after the option [8].

For entity abstraction, GSN provides the notions “Uninstantiated (UI)”, and “Uninstantiated and Undeveloped (UU)”. The former refers to abstract elements whose parameters are replaced with concrete values upon instantiation. The latter refers to UI

entities that are also undeveloped². Thus, upon instantiation, an abstract UU entity is replaced with a concrete, but undeveloped, instance.

In addition to these, there are (limited) examples of the use of a *recursion* abstraction in the literature [12], although it is not formally part of the GSN standard. Recursion, in the context of patterns, expresses the notion that a pattern (or a part of it) can itself be repeated and unrolled, e.g., as part of an optional relation or a larger pattern. Recursion abstractions may or may not be labeled with an expression giving the number of iterations to be applied in a concrete instance.

3 Formalization

In this section, first we modify an earlier definition of a partial safety case argument structure [3], [7], adding a labeling function for node contents. Then, we give a formal definition of a pattern, clarifying conditions on multiplicity and recursion. Subsequently, we give a formal semantics to patterns as the set of their concrete instances, via a notion of pattern refinement.

Definition 1 (Partial Safety Case Argument Structure). *Let $\{s, g, e, a, j, c\}$ be the node types strategy, goal, evidence, assumption, justification, and context respectively. A partial safety case argument structure \mathcal{S} is a tuple $\langle N, l, t, \rightarrow \rangle$, comprising the set of nodes, N , the labeling functions $l : N \rightarrow \{s, g, e, a, j, c\}$ that gives the node type, $t : N \rightarrow E$ giving the node contents, where E is a set of expressions, and the connector relation, $\rightarrow : \langle N, N \rangle$, which is defined on nodes. We define the transitive closure, $\rightarrow^* : \langle N, N \rangle$, in the usual way. We require the connector relation to form a finite directed acyclic graph (DAG) with the operation $isroot_N(r)$ checking if the node r is a root in the DAG³. Furthermore, the following structural conditions must be met:*

- (1) *Each root of the partial safety case is a goal: $isroot_N(r) \Rightarrow l(r) = g$*
- (2) *Connectors only leave strategies or goals: $n \rightarrow m \Rightarrow l(n) \in \{s, g\}$*
- (3) *Goals cannot connect to other goals: $(n \rightarrow m) \wedge [l(n) = g] \Rightarrow l(m) \in \{s, e, a, j, c\}$*
- (4) *Strategies cannot connect to other strategies or evidence: $(n \rightarrow m) \wedge [l(n) = s] \Rightarrow l(m) \in \{g, a, j, c\}$*

For this paper, note that in Definition 1 we have excluded the concept of an undeveloped node; consequently our definition of a pattern (Definition 2) excludes the notions of UI or UU nodes. Extending both definitions to include these is straightforward.

Definition 2 (Pattern). *A pattern \mathcal{P} is a tuple $\langle N, l, t, p, m, c, \rightarrow \rangle$, where $\langle N, \rightarrow \rangle$ is a directed hypergraph⁴ in which each hyperedge has a single source and possibly multiple targets, the structural conditions from Definition 1 hold, and l, t, p, m , and c are labeling functions, given as follows:*

² Annotating an entity as “undeveloped” is part of the main GSN syntax to indicate incompleteness, i.e., that an entity requires further support.

³ A safety case argument structure has a single root.

⁴ A graph where edges connect multiple vertices.

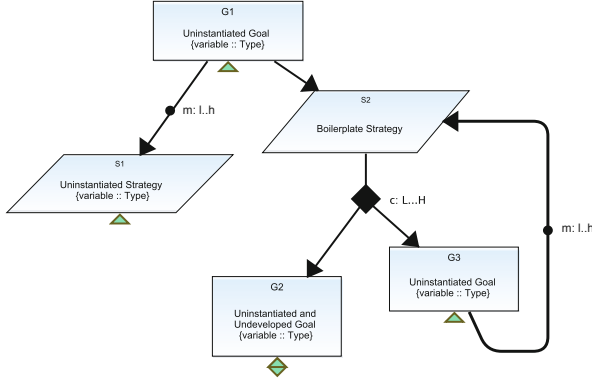


Fig. 1. Abstractions in GSN for patterns specification and our proposed modifications

- (1) l and t are as in Definition 1 above
- (2) p is a parameter label on nodes, $p : N \rightarrow Id \times T$, giving the parameter identifier and type. Without loss of generality, we assume that nodes have at most a single parameter
- (3) $m : (\rightarrow) \times N \rightarrow (N \times N)$ gives the label on the i^{th} outgoing connector⁵. Without loss of generality, we assume that multiplicity only applies to outgoing connectors. If it is $\langle l, h \rangle$ then multiplicity has the range $l..h$, where $l \leq h$. An optional connector has range $0..1$.
- (4) $c : (\rightarrow) \rightarrow N \times N$, gives the “ $l..h$ of n ” choice range. We give ranges and omit the n .

Fig. 1 illustrates the GSN abstractions for pattern specification formalized in Definition 2. We now give some notational conventions and auxiliary definitions that we will make use of:

- (a) As shown in Fig. 1, pattern nodes take parameters, which reference a set of values V , partitioned into types, and T ranges over types. We write $v :: T$, when v is a value of type T .
- (b) A pattern node N is a *data node*, written as $data(N)$, if it has a parameter, i.e., $N \in dom(p)$ (nodes G1, S1, G2 and G3 in Fig. 1). Otherwise, a node is *boilerplate* (node S2 in Fig. 1). We will write $bp(N)$ when N is a boilerplate node. For certain nodes, e.g., so-called *evidence assertions* [14], data may not be available until *after* instantiation. Although, strictly speaking, they are data nodes, we consider them to be boilerplate here (see Section 5 for an example).
- (c) The links of the hypergraph, $A \rightarrow \mathbf{B}$, where A is a single node and \mathbf{B} is a set of nodes, represent choices. We write $A \rightarrow B$ when $A \rightarrow \mathbf{B}$ and $B \in \mathbf{B}$.
- (d) The bounds on multiplicity and optionality are represented as ranges. To define the labeling functions m and c , we treat \rightarrow as a set with members $\langle A, \mathbf{B} \rangle$, where $A \rightarrow \mathbf{B}$. Then,
 - If $c(\langle A, \mathbf{B} \rangle) = \langle l, h \rangle$ we write $A \rightarrow^{l..h} \mathbf{B}$ (range on choice).

⁵ Although siblings are unordered in GSN, it is convenient to assume an ordering.

- If $m(\langle A, B \rangle, _) = \langle l, h \rangle$, we write $A \rightarrow^{l..h} B$ (range on multiplicity).
- (e) We write $sub(\mathcal{P}, A)$ for the *sub-pattern* of \mathcal{P} at A , i.e., the restriction of \mathcal{P} to $N' = \{X \mid A \rightarrow^* X\}$, and $sub(\mathcal{P}, A, B)$ for the restriction of \mathcal{P} to $\{X \mid A \rightarrow^* X \text{ and } B \not\rightarrow^* X\}$. Roughly, this is the fragment of \mathcal{P} between A and B (including A , but excluding B and everything below it).
- (f) Write $multi(\mathcal{P}, B)$ if there exists an $A \in \mathcal{P}$ such that $A \rightarrow^{l..h} B$ and $h > 1$, that is, pattern node B can be repeated in instances of \mathcal{P} . We will write $multi(B)$ when \mathcal{P} is obvious, and often consider $multi(G, B)$, where G is a subgraph of \mathcal{P} .
- (g) A path, s , in the pattern is a sequence of connected nodes. If s connects A and B , we write this as $s : A \rightarrow^* B$.
- (h) Write $A < B$ if for all paths from the root $s : R \rightarrow^* B$, we have $A \in s$.
- (i) Write $A \rightarrow^n B$ when there is a path of length n in the pattern between nodes A and B . Then we define $A \rightarrow^{must} \mathbf{B}$, when every path from A that is sufficiently long must eventually pass through some $B \in \mathbf{B}$, i.e., $\exists n. \forall s : A \rightarrow^n X. \exists B \in \mathbf{B}. B \in s$.

We now introduce a restriction on the combination of multiplicities and boilerplate nodes. The intuition is that multiplicities should be resolved by data, and not arbitrarily duplicated: it is only meaningful to repeat those boilerplate nodes associated with distinctly instantiated data nodes.

Definition 3 (Multiplicity Condition). *We say that a pattern satisfies the multiplicity condition when for all nodes B , if $multi(B)$, and not $data(B)$, then there exists a C such that $B \rightarrow^* C$, $data(C)$, and for all X such that $B \rightarrow^+ X \rightarrow^* C$, not both $multi(X)$ and $bp(X)$.*

In other words, a multiplicity that is followed by boilerplate must eventually be followed by a data node, with no other multiplicity in between. This has two consequences: (i) we cannot have multiplicities that do not end in data, and (ii) two multiplicities must have intervening data.

In contrast to concrete argument structures, we allow cyclic structures and multiple parents in patterns. However, we need a restriction to rule out ‘inescapable’ loops, so that recursion is well-founded.

Definition 4 (Well-foundedness). *We say that an argument pattern is well-founded when, for all pattern nodes A , and sets of nodes \mathbf{B} , such that $A \notin \mathbf{B}$, if $A \rightarrow^{must} \mathbf{B}$ then it is not the case that for all $B \in \mathbf{B}$, $B \rightarrow^{must} A$.*

We give semantics to patterns in the style of a single-step refinement relation \sqsubseteq_1 . Intuitively, the idea is to define the various ways in which indeterminism can be resolved in a pattern. As before (Definition 2), pattern $\mathcal{P} = \langle N, l, t, p, m, c, \rightarrow \rangle$ and we describe the components of \mathcal{P} which are replaced in \mathcal{P}' .

Definition 5 (Pattern Refinement). *For patterns $\mathcal{P}, \mathcal{P}'$, we say that $\mathcal{P} \sqsubseteq_1 \mathcal{P}'$ iff any of the following cases hold:*

- (1) *Instantiate parameters: If $p(n) = \langle id, T \rangle$ and $v :: T$, then replace node contents, t , with $t' = t \oplus \{n \mapsto t(n)[v/id]\}$.*
- (2) *Resolve choices: If $A \rightarrow^{l..h} \mathbf{B}$, $\mathbf{B}' \subseteq \mathbf{B}$ and $l \leq |\mathbf{B}'| \leq h$, then replace $A \rightarrow \mathbf{B}$ with $A \rightarrow \mathbf{B}'$ for each $B \in \mathbf{B}'$.*

- (3) Resolve multiplicities: If $A \rightarrow^{l..h} B$ then replace the link $A \rightarrow B$ with n copies (that is, disjoint nodes, with the same connections and labels), where $l \leq n \leq h$.
- (4) Unfold loops: If $A \rightarrow^* B$, $B \rightarrow A$, and $A < B$, then let S be the sub-pattern of \mathcal{P} at A , $\text{sub}(\mathcal{P}, A)$. We create a copy of S and replace the link from B to A with a link from B to the copy of S (i.e., we sequentially compose the two fragments).

Then, $\mathcal{P} \sqsubseteq \mathcal{P}'$ iff $\mathcal{P} \sqsubseteq_1^* \mathcal{P}'$.

We will define pattern semantics in terms of refinement to arguments. Formally, however, a pattern refines to another pattern, so we need to set up a correspondence between concrete patterns and arguments structures. We define this as an embedding from the set of argument structures into patterns.

Definition 6 (Pattern Embedding). An embedding \mathcal{E} of an argument structure into a pattern is given as $\mathcal{E}(\langle N, l, t, \rightarrow \rangle) = \langle N, l, t, p, m, c, \rightarrow' \rangle$ where $p = \emptyset$, the labeling functions m and c always return 1..1, and hyperedges have a single target, i.e., for all nodes $A \in N$, $\rightarrow'(a) = \{\rightarrow(a)\}$.

We can now define the semantics of a pattern as the set of arguments equivalent to the refinements of the pattern.

Definition 7 (Pattern Semantics). Let \mathcal{P} be a pattern, let C and A range over patterns, and safety case argument structures, respectively. Then⁶, we give the semantics of \mathcal{P} as $\llbracket \mathcal{P} \rrbracket = \{A \mid \mathcal{P} \sqsubseteq C, \mathcal{E}(A) = C\}$.

4 Instantiation

Now, we formalize the concept of a pattern dataset, define a notion of *compliance* between data and a pattern, and specify a generic instantiation algorithm.

4.1 Datasets and Tables

We use sets of values to instantiate parameters in patterns to create instance arguments. Roughly speaking, data can be given as a mapping from the identifiers of data nodes to lists of values. However, since a pattern is a graph there can be multiple ways to navigate through it (due to recursion and nodes with multiple parents) and, therefore, connect the instance nodes. To make clear where an instantiated node should be connected, we need to associate each ‘instantiation path’ through the pattern with a *join point* (or simply *join*), indicating where a ‘pass’ through the pattern begins. A join uniquely indicates the location at which an instantiated branch of the argument structure is to be appended. In practice, join points can be omitted if the location can be unambiguously determined.

We adopt a liberal notion of pattern instance and do not require all node parameters to be instantiated. Moreover, uninstantiated nodes do not appear in the resulting instance⁷.

⁶ Strictly speaking, this should be defined as a set of equivalence classes of arguments, where we abstract over node identifiers, but we can safely gloss over that here.

⁷ Except for special cases where they have been considered as boilerplate (see item (b) on p. 24).

Definition 8 (Pattern Dataset). Given a pattern, \mathcal{P} , define a \mathcal{P} -dataset as a partial function $\tau : (D \times V) \times D \rightarrow (\mathbb{N}^* \rightarrow V)$, where D is the set of data nodes in \mathcal{P} , V is the set of values, and \mathbb{N}^* is the set of indices. We write $v \in^{r,c} \tau$ when for some i , $\tau(r, c)(i) = v$, and require that values be well-typed, i.e., if $v \in^{r,c} \tau$ and $p(c) = \langle id, T \rangle$ then $v :: T$.

Data will typically be represented in tabular form where we label columns by data nodes, D , and rows by $D \times V$ pairs, i.e., joins. Entries in the table are represented as indexed lists of values. The order in which a dataset is tabulated does not actually provide any additional information, but in order to be processed by the instantiation algorithm, must be *consistent* with the pattern, in the following sense: the order of columns must respect node order⁸ $<$, i.e., if $A < B$ then the corresponding columns are in that order; and for each row $\langle D, v \rangle$, we require that v appears in column D in a preceding row. In the following, we will assume that a consistent order has been chosen for a dataset, and refer to it as a \mathcal{P} -table (see Table 1 for an example).

Definition 9 (Data Compliance). For pattern \mathcal{P} and \mathcal{P} -table τ , we say that the table complies with the pattern, $\tau \models \mathcal{P}$, if the following two conditions hold:

- (i) τ meets the cardinality constraints of \mathcal{P} , i.e., $\forall c. l \leq |\tau((-, -), c)| \leq h$, where $\langle l, h \rangle = m(i, c')$, where $c' \rightarrow^i c$.
- (ii) τ is upwards-closed, i.e., for each r labeled (D, v) and column c , if $v \in^{r,c} \tau$ and $c \leq c' \leq D$, then there exists v' such that $v' \in^{r,c'} \tau$.

Note that the ordering used in upwards-closure is with respect to the pattern and not the column order. The intuition behind upwards closure is that, in line with our notion of partial instantiation and although not all nodes need be instantiated, we do require that parameters can be instantiated in order from the root. A row, therefore, consists of the data that instantiates an *upward-closed* fragment of the pattern, following the paths of the fragment up until the join (see Fig. 4 for an example).

4.2 Algorithm

Fig. 2 specifies *instantiate*(\mathcal{P}, τ), our generic algorithm for pattern instantiation. We write *new*($D.v$), to create a new instance node, given by instantiating data node D with value v . When a boilerplate node B is instantiated, then we reference its instance simply as B . Let \mathcal{F} be the set of argument structure fragments. To connect an instance node $D.v$ to a fragment $f \in \mathcal{F}$, we use a function *connect*($D.v, f$), which sequentially composes f with the current instance fragment at node $D.v$.

To instantiate a pattern \mathcal{P} , given its \mathcal{P} -table τ , we process each row to create a *row instance* fragment, which is effectively the assignment of parameter values in the table to the corresponding data nodes in the pattern. We construct the row instance based on the ordering of the data nodes in the columns. For each value we add not just the instantiation of the appropriate data node, but also any boilerplate between that node and the preceding data node. We give a row instance as $RI \in N \times \mathbb{N} \times \mathbb{N}^*$, where N , \mathbb{N} , and \mathbb{N}^* are the sets of pattern nodes, instance nodes and natural number indices respectively.

⁸ See item (h) on p. 25.

```

1 Instantiate( $\mathcal{P}$ : Pattern,  $\tau$ :  $\mathcal{P}$ -table)
2 begin
3   foreach row  $r \in$  table  $\tau$  do
4     initialize row instance  $RI \leftarrow \emptyset$ 
5     if row label =  $\langle root, v \rangle$  then
6       create instance node  $j \leftarrow new(root.v)$  and assign pattern node  $current \leftarrow root$ 
7       update row instance  $RI \leftarrow RI \cup \langle current, j, [] \rangle$ 
8     else if row label =  $\langle C, v \rangle$  then
9       create join instance node  $j \leftarrow new(C.v)$  and assign  $current \leftarrow C$ 
10    foreach column  $E \in$  table  $\tau$  not including root do
11      assign pattern node  $N \leftarrow current$ 
12      foreach ( $v, index\ i \in$  table  $\tau(r, E)$  do
13        assign fragment  $f \leftarrow$  boilerplate  $B \in sub(\mathcal{P}, current, E)$  such that  $multi(B) \vee \langle B, B, [] \rangle \notin RI$ 
14        if  $E$  is first column in row  $r$  with data then assign instance node  $n \leftarrow j$ 
15        else find parent node  $n$  with index  $k$  such that  $\exists \langle N, n, k \rangle \in RI$ 
16        connect( $n, f$ )
17        foreach boilerplate  $B \in f$  do update row instance  $RI \leftarrow RI \cup \langle B, B, i \rangle$ 
18        if  $\exists P \in sub(\mathcal{P}, current, E) \mid multi(P)$  then assign pattern node  $N \leftarrow parent(P)$ 
19        assign pattern node  $M \leftarrow parent(E)$ 
20        assign instance node  $p \leftarrow$  instance node  $m \in f$  such that  $m = M.v$ 
21        connect( $p, E.v$ )
22        update row instance  $RI \leftarrow RI \cup \langle E, E.v, i \rangle$ 
23        assign  $current \leftarrow E$ 

```

Fig. 2. Generic algorithm for pattern instantiation

Multiplicities, especially, require careful consideration: multiple values in the \mathcal{P} -table lead to multiple instances of a data node, but we only repeat those boilerplate nodes which appear after a multiplicity (see Fig. 4 for an example). We use instance indices to connect nodes to the correct parent when there are such multiples. At any point in the algorithm we identify the “current node” as *current*, and the pattern root as *root*.

We now state, without proof, the correctness property of the instantiation algorithm.

Correctness: If \mathcal{P} is a well-founded pattern that satisfies the multiplicity condition, and $\tau \models \mathcal{P}$, then $instantiate(\mathcal{P}, \tau) \in \llbracket \mathcal{P} \rrbracket$. A consequence is that the algorithm produces well-formed instances.

5 Illustrative Example

To illustrate pattern instantiation, we use the *requirements breakdown pattern* (Fig. 3), which we have derived from our ongoing experience with safety case development for an unmanned aircraft system [2], [4], [6]. It also extends our previous work⁹ on algorithmically deriving argument structure fragments from requirements/hazards tables [3].

The requirements breakdown pattern (Fig. 3) provides a framework to abstractly represent the argument implicit in a requirements table¹⁰. Specifically, it shows how the claims entailed by requirements can be hierarchically developed and linked to the supporting evidence produced from the specified verification methods. Due to space limitations, we do not provide a complete pattern specification.

⁹ In fact, a \mathcal{P} -table similar to Table 1 can be extracted from the tables in [3].

¹⁰ See [3] for an example of a requirements table.

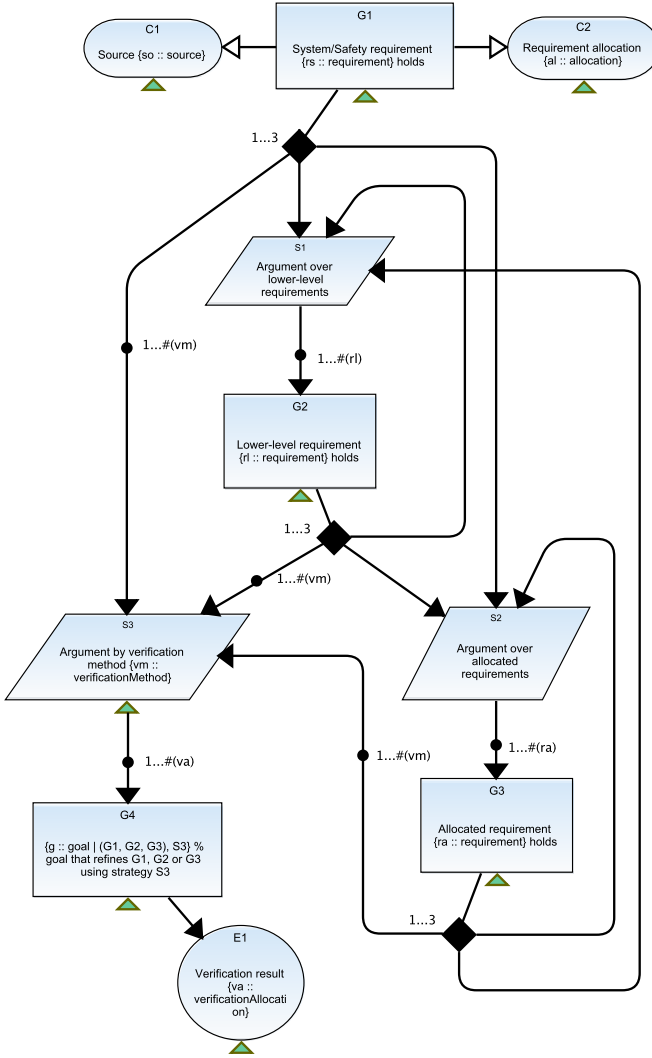


Fig. 3. Requirements breakdown pattern, abstracting the structure of the argument implicit in a requirements table

Table 1. Example of a populated \mathcal{P} -table to instantiate the requirements breakdown pattern

Parameter Type	Requirement	Lower-level requirement	Allocated Requirement	Source	Requirement Allocation	Verification Method	Verification Allocation
Data node	G1	G2	G3	C1	C2	S3	E1
Join	R1	R1.1, R1.2	AR1	S	A	VM11, VM12	VA11, VA12
	(S3, VM12)						VA22
	(G2, R1.1)					VM1.11, VM1.12	VA1.11, VA1.12
	(G2, R1.2)	R1.2.1, R1.2.2	AR1.2				
	(G2, R1.2.1)					VM1.2.1	VA1.2.1
	(G3, AR1.2)		AR1.21			VM1.2	VA1.2

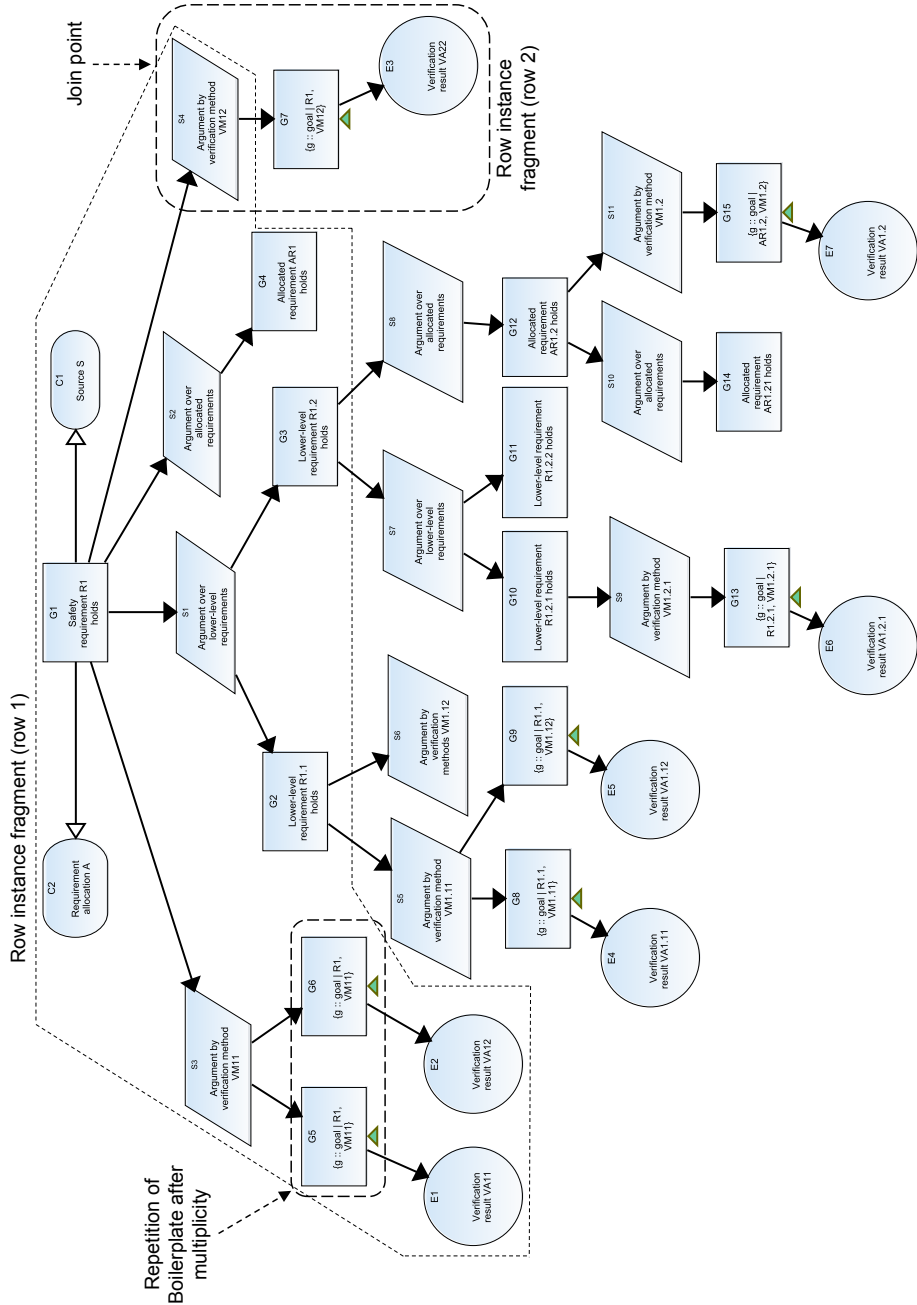


Fig. 4. Application of the generic pattern instantiation procedure (Fig. 2): Concrete instance of the requirements breakdown pattern (Fig. 3) using the values from the \mathcal{P} -table (Table 1), highlighting row instance fragments, join points and repetition of boilerplate nodes

In brief, the claim in the root goal (G1) of the pattern is that a safety/system requirement, which is usually made in the contexts of some source (C1), or system, i.e., requirement allocation (C2), holds. A choice of three strategies is available to develop G1: hierarchical decomposition (S1, S2) and appeal to one or more verification methods (S3). The sub-claims (G2, G3) resulting from applying hierarchical decomposition are semantically similar to the root claim that they refine. Consequently, we can apply the same strategies to develop them further. Eventually, we support all claims by verification evidence (E1). The evidence is preceded by an *evidence assertion* (G4), i.e., a minimal proposition directly concerning the source data of the evidence [14].

Table 1 shows a populated \mathcal{P} -table for the requirements breakdown pattern with the columns, labeled by the pattern data nodes, containing example data entries entered corresponding to the root node and the join points. We have listed the data node parameter type for clarification purposes and it is not formally part of the data model.

Fig. 4 shows an instance of the pattern derived by applying our generic pattern instantiation procedure (Algorithm 2) and using the \mathcal{P} -table (Table 1). It highlights the repetition of boilerplate nodes¹¹ after multiplicity, and illustrates how a join point connects two row instance fragments.

6 Conclusion

We have presented the foundational steps towards, we believe, a rich theory of safety case patterns that will enable more sophistication in their usage than is currently available, e.g., automated instantiation, composition, and transformation-based manipulation. The main benefit of our work from a practitioner’s perspective, we anticipate, is a reduction in the effort involved in safety case creation/management due to the raised level of abstraction at which arguments can be formulated, together with improved assurance. Specifically, given the assurance afforded by automated instantiation that a pattern instance is well-formed and meets its specification, practitioners, i.e., safety engineers who create safety arguments, and certification/qualification authorities who evaluate them, can divert efforts to domain-specific issues, e.g., selecting the appropriate patterns for assurance, evaluating a smaller, abstract argument structure for fallacies/deficits instead of its larger concrete instantiation, determining the evidence required to support the claims made, etc.

However, more can be done: as mentioned earlier, the formal definitions and the algorithm can be extended to include the notions of undeveloped, UI and UU. One design choice in the algorithm was to instantiate only those nodes for which parameters have values in the data table. An alternative choice could be to use the whole pattern so that those data nodes that do not take values in the table are also reproduced in the instance but left as UI or UU, as appropriate. The relationship between modular abstractions, hierarchies [7], and patterns is, as yet, unclear although there are a few examples of applying patterns within a modular organization [9]. The goal of formalization, here, would be to raise the level of abstraction and to increase automation. We use a notion of sequential composition of patterns. We have also defined a notion of parallel composition (not given in this paper) to create patterns, such as for requirements breakdown

¹¹ Recall that we consider evidence assertion nodes as boilerplate (see item (b) on p. 24).

(Fig. 3), from simpler patterns. Future work will involve, in part, extending the formal basis given in this paper to the topics mentioned above.

We have already implemented the GSN abstractions and our notational extensions for patterns in our toolset, AdvCATE [5]; we plan to extend the tool with the algorithm described here. Clarifying concepts such as patterns and the data for their instantiation will be necessary to support tool interoperability, which is one of the goals [13] of emerging safety/assurance case standards.

Acknowledgement. This work has been funded by the AFCS element of the SSAT project in the Aviation Safety Program of the NASA Aeronautics Mission Directorate.

References

1. Alexander, R., Kelly, T., Kurd, Z., McDermid, J.: Safety Cases for Advanced Control Software: Safety Case Patterns. Final Report, NASA Contract FA8655-07-1-3025, Univ. of York (October 2007)
2. Denney, E., Habli, I., Pai, G.: Perspectives on Software Safety Case Development for Unmanned Aircraft. In: Proc. 42nd IEEE/IFIP Intl. Conf. Dep. Sys. and Networks (June 2012)
3. Denney, E., Pai, G.: A lightweight methodology for safety case assembly. In: Ortmeier, F., Lipaczewski, M. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 1–12. Springer, Heidelberg (2012)
4. Denney, E., Pai, G., Pohl, J.: Automating the generation of heterogeneous aviation safety cases. Tech. Rep. NASA/CR-2011-215983, NASA Ames Research Center (August 2011)
5. Denney, E., Pai, G., Pohl, J.: AdvCATE: An Assurance Case Automation Toolset. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP Workshops 2012. LNCS, vol. 7613, pp. 8–21. Springer, Heidelberg (2012)
6. Denney, E., Pai, G., Pohl, J.: Heterogeneous aviation safety cases: Integrating the formal and the non-formal. In: 17th IEEE Intl. Conf. Engineering of Complex Computer Systems pp. 199–208 (July 2012)
7. Denney, E., Pai, G., Whiteside, I.: Hierarchical safety cases. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 478–483. Springer, Heidelberg (2013)
8. Goal Structuring Notation Working Group: GSN Community Standard Version 1 (November 2011), <http://www.goalstructuringnotation.info/>
9. Industrial Avionics Working Group: Modular Software Safety Case Process, Parts A and B: Process and Guidance. Tech. Rep. IAWG-AJT-301, Issue 2 (October 2007)
10. Kelly, T.: Arguing Safety: A Systematic Approach to Managing Safety Cases. Ph.D. thesis, Univ. of York (1998)
11. Kelly, T., McDermid, J.: Safety case construction and reuse using patterns. In: Daniel, P. (ed.) Safe Comp 1997, pp. 55–69 (1997)
12. Menon, C., Hawkins, R., McDermid, J.: Interim standard of best practice on software in the context of DS 00-56 Issue 4. SSEI Standard of Best Practice (Issue 1). Univ. of York (2009)
13. Object Management Group: Structured Assurance Case Metamodel (SACM) version 1.0. Formal/2013-02-01 (February 2013), <http://www.omg.org/spec/SACM/>
14. Sun, L., Kelly, T.: Elaborating the concept of evidence in Safety Cases. In: Proc. 21st Safety Critical Sys. Symp. (February 2013)
15. Weaver, R.: The Safety of Software – Constructing and Assuring Arguments. Ph.D. thesis, Dept. of Comp. Sci., Univ. of York (2003)

Testing Autonomous Robot Control Software Using Procedural Content Generation

James Arnold¹ and Rob Alexander²

¹ BAE Systems Detica, UK

james.arnold@detica.com

² Department of Computer Science, University of York, UK

rob.alexander@york.ac.uk

Abstract. We present a novel approach for reducing manual effort when testing autonomous robot control algorithms. We use procedural content generation, as developed for the film and video game industries, to create a diverse range of test situations. We execute these in the Player/Stage robot simulator and automatically rate them for their safety significance using an event-based scoring system. Situations exhibiting dangerous behaviour will score highly, and are thus flagged for the attention of a safety engineer. This process removes the time-consuming tasks of hand-crafting and monitoring situations while testing an autonomous robot control algorithm. We present a case study of the proposed approach – we generated 500 randomised situations, and our prototype tool simulated and rated them. We have analysed the three highest rated situations in depth, and this analysis revealed weaknesses in the smoothed nearness-diagram control algorithm.

Keywords: autonomy, robots, faults, simulation, procedural content generation.

1 Introduction

There is much work afoot to bring autonomous robots (AR) into public spaces, both on the ground and in the air. We therefore need high confidence that their behaviour will be safe. This is difficult, however, given the complexity of environments that the robots must interact with, and the scope of authority that they need in order to effectively respond to those environments. There have already been minor accidents caused by the control software of autonomous vehicles, for example in the DARPA Urban Challenge [1], and there are likely to be more as their numbers increase. We need ways to verify and validate such control software, finding ways in which it could cause an accident, whether due to intrinsic algorithmic limitations or implementation-specific faults.

One approach would be testing AR by putting them through linear scenarios, represented as a sequence of stimuli, and checking the safety of the resulting behaviour. This is inadequate, however: AR will influence their environment, and that will influence their own future behaviour. This is true, of course, for any reactive system, but AR may have many ways to influence their environment (e.g. movement,

manipulator arms, communication) and many ways to respond to those changes (through sensing, model-building and prediction). We thus need to understand how the behaviour of the AR will develop from a given starting point. We therefore need to test AR by generating *situations* – combinations of maps, peer entities, and missions or objectives, into which a (simulated) AR can be placed. Rather than providing a fixed sequence of stimuli, simulated situations can react to the AR's actions. Given such a situation, we can assess whether the AR achieves safe behaviour.

For example, one situation might have an autonomous ground vehicle driving down an empty motorway to deliver a package. Another might modify the mission to have a tight time limit and stationary traffic jam. A third situation might instead pepper the motorway with abandoned cars and spilt fuel. Each of these would test different aspects of the vehicle's control software.

We can observe that a situation-based approach can provide a degree of both verification (of the implementation against a vehicle behaviour specification) and validation (of the behaviour specification against the behaviour that we, in retrospect, want from it, now that we've seen the consequences).

There are two major obstacles for situation-based testing. First, useful situation-based testing will require a wide variety of situations, and creating those in a simulation requires considerable effort – for example, engineers need to list the precise position, heading and plan of each vehicle involved in each situation. This involves a lot of work, perhaps engineer-weeks for a complex situation, and this thus limits the range of situations that can be studied. Martin and Hughes report similar problems with scenarios for training simulations [2].

Second, the *diversity* of generated situations is important. In particular, we do not want biases in the generation of situations that mirror the biases of the designers of the control algorithms; we do not want our test set to have the same 'blind spots' as the software. Research on n-version programming suggests that this is likely – in Knight and Leveson's classic experiment [3], different programmers working from the same specification produced programs that had largely similar errors. It may be that different situation designers produce situation sets which miss similar challenges.

A response to these two problems is to generate situations automatically. This is the response taken in the simulation-for-training field, where it is called Automatic Situation Generation (ASG) e.g. Martin and Hughes [2] and in the video games field, where it is called Procedural Content Generation (PCG) e.g. Togelius et al [4, 5]. PCG is often used in situations where the manual creation of content would be prohibitively time-consuming or expensive [6]. It also allows highly detailed content to be stored extremely efficiently—only the parameter values input into the algorithms need to be stored. PCG has the potential to produce output that surprises the PCG engine developers; this has often been observed in video game uses [7].

In this paper we show how PCG techniques can be adapted to test AR control software, and demonstrate this by application to a small case study. In section 2 we identify the requirements for such an approach, in section 3 we explain what we have implemented, and in section 4 we describe an initial experimental evaluation. Section 5 compares our approach against previous work; section 6 summarises the paper and identifies future possibilities.

2 Requirements for a PCG Testing Approach

The major criterion for evaluating a situation-based testing approach is its ability to find the kind of problem behaviours that AR may exhibit. Here, for simplicity, we have limited our scope; we only consider how the combination of sensing, sensor processing and driving algorithms can lead to dangerous movement behaviour. To support this, our situation generator needs the ability to generate static terrain, including plausible variations of terrain type such as rocky deserts, cave systems, and urban areas. It also needs to generate moving obstacles, such as peer robots, as these are particularly difficult to sense and respond to appropriately.

A key requirement is that the PCG technique generates a *wide range of diverse situations*. Beyond mere diversity, it is also important to generate situations that have a *high likelihood of finding dangerous behaviour*; it would be easy for PCG to spend most of its time generating safe situations, which is computationally inefficient.

Regardless of the efficiency of the generator, an effective PCG technique will generate a great many situations, necessitating a great many simulation runs, and hence generating a very large amount of data. We thus need a way to rationalise the output before it is presented to human engineers. At a minimum, this must filter out runs that are safe and as expected. Beyond that, the tool must prioritise the runs in some way, allowing the humans to focus first on the most interesting ones.

Ideally, we would want to gain confidence that the set of situations generated is in some sense ‘complete’, at least for the scope of situations being considered. This could be approached by measuring the “situation coverage” achieved by a set of situations – given the range of situations that could potentially be generated, the proportion that actually have been generated. This could be composed from a variety of components such as potential for pair-wise interactions between entity types, where entities include (for example) the AR, peer vehicles, roads, walls, and solid obstacles, and interactions include (for example) sensing, proximity, collision avoidance and “moving in formation with”. Previous work on agent interaction ontologies (e.g. Nguyen [8]) and on road-traffic interaction patterns and accident proximity measures (e.g. Archer [9]) will be relevant here.

Given the proof-of-concept nature of the work described here, we have not attempted to address the completeness issue. Instead, we have limited ourselves to the assertion that if the method discovers any specification or implementation flaws then it has some value. The decision to use it then becomes a return-on-investment question.

3 Proposed Method

The basic idea is to use PCG to create situations, and then run them in the simulator to observe how the robot (specifically, its control software in command of the robot) behaves in them. The process takes place in three stages.

The first stage uses PCG algorithms to create a binary terrain map, i.e., terrain is either navigable or obstructed. A mission generator then reads the map and places a

number of robots within the environment, ensuring they are initially unobstructed. It then assigns each robot a randomised route, which the robot control algorithm must follow during the second stage. The output of the first stage is a complete situation: a fully-populated environment with robots, obstructions and mission allocations.

The second stage involves executing a situation within a simulated environment. During its execution, the progress and behaviour of the robots within the simulation are monitored to determine whether any hazardous behaviour is exhibited. Monitoring is done by a daemon process which interfaces with the simulation. The daemon inspects the state of the robots to detect whenever an event occurs that corresponds to a member of a defined set of unwanted behaviours, such as colliding with a wall. The occurrence of such an event would indicate a failure of the control algorithm.

Once a run is complete, a third stage processes the event log produced by the daemon. The processing analyses the sequence of events within the run and ranks its overall significance, based on the likelihood that faults or weaknesses of the robot control algorithm were exhibited.

The output of the third stage is a ranked list of situations, ordered by their significance. A domain expert can use this to indicate which situations are most likely to be worth investing further time and resources in, using more labour-intensive techniques.

In the current work, we used Player/Stage [10] as our implementation platform. Player is an abstraction library that provides a standardised interface to robot sensor and actuator hardware. Player also features a client/server architecture, allowing control code to execute across a network connection, rather than only running on-board a robot. It is widely used by robotics researchers as it enables a seamless transition between simulation and physical hardware. This allows developers to, for example, identify a weakness in a control algorithm in simulation, then directly test it in the real world to verify that there is really a problem.

Stage is an open-source robot simulation environment that allows one or more robots to explore and interact within a 2D world. It is often used in conjunction with the Player project, as it provides virtualised hardware for many of the interfaces Player defines. Stage accurately models a range of sensors often found on physical hardware, such as laser rangefinders, and it simulates realistic physics with accurate collision detection. The limitation to a 2D world is pertinent, but is not a major limitation for this proof-of-concept study, and Player is fully compatible with 3D simulations including Gazebo (<http://www.gazebo.org/>).

To generate an environment representing terrain and built structures the tool first creates a 2D noise map using a Perlin noise process [11]. As with most PCG techniques, this is used as a base for subsequent post-processing. Using a simple pipe-and-filter architecture, filter effects can be applied and chained arbitrarily to produce the final output that represents terrain. In the current tool we use two filters: a pixelisation filter to make the noise more granular and a thresholding filter to convert the noise into binary occupied/unoccupied. The latter is wrapped by a coverage constraint algorithm that randomly varies the threshold used by the filter until the coverage of the space reaches a suitable mix of occupied and unoccupied space (as defined in a parameter file). Fig. 1 shows the effects of the filters.

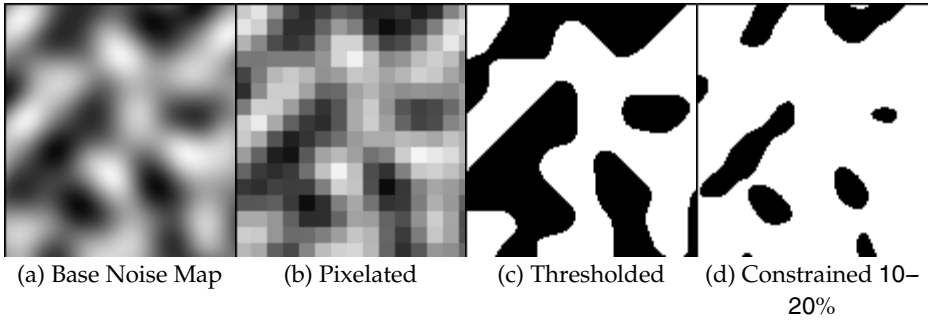


Fig. 1. Effects of post-processing filters available during environment generation

The tool is configured by a situation file, which specifies various parameters to use during generation, such as the size of the map and the settings for the filters. The combination of these parameters and random variation gives rise to several different types of environment – examples are given in Fig. 2.

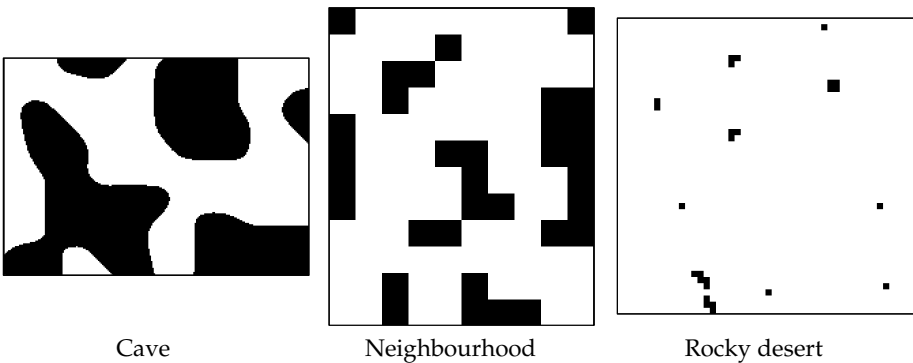


Fig. 2. Range of possible environments (with suggestive names)

The mission planning tool takes as input the map generated by the environment generator and a second configuration file. The configuration file specifies the number of robots that need missions, including any (optional) minimal route length constraints. The mission planner then places each robot within the environment, ensuring they are not wholly or partially within an obstruction. It then generates a randomised route, defined by an ordered sequence of waypoints, for each robot.

To generate a route, the planner picks an unobstructed point at random, then applies the A* path-finding algorithm [12] between a robot's initial position and the selected point. If no navigable route can be found then the end point is discarded and a new one selected. If a minimum route length is specified in the configuration file, additional end points are appended until the constraint is satisfied. In practice, this leads to routes that closely traced the edges of terrain features, which is not sensible given that the robot needs to maintain useful lines of sight. For the purpose of route planning, therefore, we therefore performed a binary dilation on the map to

exaggerate the size of terrain objects and thus force the routes to be plotted through more open space.

The pathfinding algorithm generates complete, navigable routes with a great many waypoints. The Ramer–Douglas–Peucker¹ algorithm is used to reduce the number of waypoints, both to reduce storage space for the situation and to force robots to do some online global path-planning to avoid obstacles, thus increasing the challenge for their movement algorithms. The algorithm’s epsilon parameter, which determines the amount by which the simplified path can deviate from the original (and thus the amount of global planning that can be required), can be set in the configuration file.

Failures are detected by a daemon that continuously monitors the simulation environment. The daemon is independent of the robot controller code and can detect loitering, route completion, stalls (which are produced by Stage when the robot collides with terrain or another robot), waypoint arrival and unsafe proximity (between two robots). These are a mixture of safety, mission and performance events; in future work we plan to expand the range of safety-specific measures. In effect, it filters the fine detail of the simulation to produce a simpler textual representation (an event log) of each run.

The logs are human readable, but very long, and one log is produced for every situation that is run. In order to guide engineers to the most interesting runs in the log, we developed a process for scoring runs according the interestingness (i.e. safety-significance) of each the log events in that run. The score for each run is derived by applying event-specific penalties each time an event is triggered.

Appropriate values to use for these penalties are subjective and highly dependent on both the features of the control algorithm being assessed and the context under which a robot will operate. The values used during the case study for this project were tuned using trial and error. For example, a penalty of 10 points is added if two robots enter an unsafe proximity, and a further penalty of 1 point per second is applied for the duration of the proximity. An idea for evolving or learning weights is presented in section 6.

We can note that any controller that this implementation was applied to would have to support the Player/Stage-supplied virtual hardware and 2D environment, but otherwise the robot software is independent of our approach and the tool implementation is completely independent of the specific control software used. It could easily be applied to alternative algorithms or rival software designs.

The full source code for our implementation is available at http://www-users.cs.york.ac.uk/~rda/arnold_sitgen_src.zip, under a BSD licence.

4 Experiment and Results

4.1 Experiment

For the case study, we implemented a simple robot controller that combined simple path-following with smoothed nearness diagrams (SNDs) [13] for collision avoidance

¹ Also known as the “split-and-merge” algorithm.

and basic reactive navigation. SNDs are well-understood and have been designed to work in both open and confined environments, including those with dynamic obstacles. This meant that they were suitable for any of the environments created by the environment generator and mission planner. SNDs are often used in robotics research, thus an implementation is bundled with the Player distribution. Finding any weaknesses or faults in that would be evidence in favour the approach taken by this project on a real world and well-tested control algorithm, and it would also provide useful feedback for the Player developers.

The controller was implemented on a virtual Pioneer 3-AT robot (<http://www.mobilerobots.com/ResearchRobots/P3AT.aspx>) equipped with a SICK LMS200 laser rangefinder; a model for this is provided by Stage. The robot is four-wheel drive, uses skid-steering and is capable of speeds of up to 0.8m/s. Although equipped with a sonar array, we used the laser range finder for obstacle detection as it gave us a forward-facing 180° field of view, a maximum sense range of 10m and precision to the nearest cm (surpassing the capabilities of the sonar array).

For the experiment, we generated and ran 500 unique situations, each with randomised parameters such as map size, obstacle density and minimum route lengths. Each situation was run as fast as Stage could simulate, before being terminated after 120 seconds of wall-clock time². Additionally, each situation contained a single AR and between zero and five ‘dumb’ robots; the latter acted as dynamic obstacles. The dumb robots had collision avoidance disabled and were only allocated routes that did not require any path-planning, i.e., adjacent waypoints could always be reached as the crow flies. Failure detection was only performed for the AR; events concerning only dumb robots did not contribute to the rating of a situation.

The source code and Player/Stage configuration files used for the case study are included in the code distribution linked earlier. Further details are given in [14].

4.2 Results

The risk scores for the different runs were widely distributed; indeed they appeared to follow a power-law distribution with a long tail of low-scoring (low-risk) runs. We investigated several of them in detail, and will discuss the highest-scoring three here. These scored 3064, 1574 and 988 respectively, versus a mean of 61.0. All of these high-scoring runs involved collisions between the AR and one of the dumb robots. The maps and initial mission plans for these runs are shown in figures 3-5. Robot starting points are shown by squares, robot goals are shown by circles; those for the AR are filled shapes, those for the dumb robot are hollow. The green lines represent the route plans provided to the robots; those for the AR have been simplified (as described in section 3) and thus cannot be completed without obstacle avoidance.

In run 207 (ranked first), a repeat collision occurred because the AR rounded a corner and collided with a dumb robot that was outside its field of view. As the dumb robot was still out of view after the collision, the AR continued to try to drive through

² The final simulation time of each run varied, but it tended to be on the order of 10 minutes, a simulation performance of around 5x real time.

it, thus continually pushing against it (which was logged as a very large number of collisions). This is a simple example of a hazard.

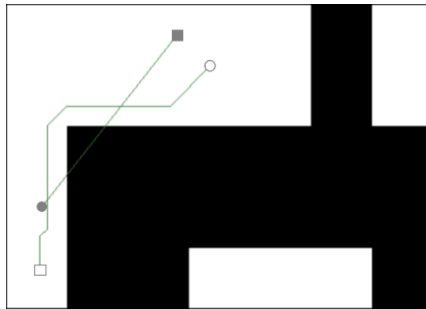


Fig. 3. Environment and mission plan for situation 207

In run 219 (ranked second), the AR and a dumb peer were started very close to each other on routes that took them through each other. There *was* space for them to avoid each other and pass successfully, but they started too close for the AR’s avoidance algorithms to work properly. The AR therefore oscillated between trying to pass to the left and trying to pass to the right, colliding repeatedly with the dumb robot.



Fig. 4. Environment and mission plan for situation 219

This is interesting because *smoothed* nearness diagrams were meant to fix the oscillation proneness of the original nearness diagram concept, but here this was defeated because the AR could not see enough free space. The problem could be solved if the AR could comprehend the problem and back up slightly, but the AR’s control system does not have that feature.

Run 231 (ranked third) was similar to run 207, except that the AR *could* see the dumb robot. It could have avoided a collision by moving at full speed, thus getting out of the way before the dumb robot reached it. SND, however, is designed to command

low speed in “high risk” areas (ones near obstacles). In this case, the obstacle was itself moving towards the SND-equipped robot so high speed was necessary to avoid it. SND guided the AR to move slowly, thus colliding with the dumb robot.

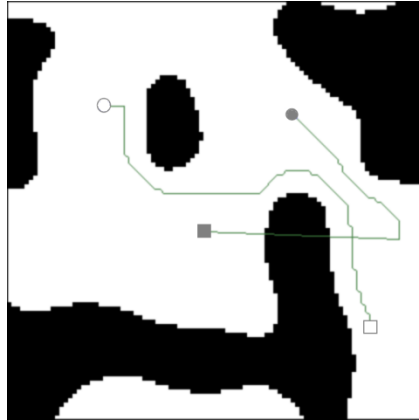


Fig. 5. Environment and mission plan for situation 231

Run 231 is perhaps the most interesting of the three runs, as it reveals a weakness in the established SND concept, not just with the robot’s perception (run 207) or with its specific control implementation (run 219). Despite this, it is only scored at one-third the value of run 207. It is therefore clear that some fine-tuning of the scoring system is desirable; with this in mind, we looked at the overall picture of whether the scoring had sorted runs into roughly the right order.

The three runs described above are examples of true positives – runs scored highly that contain accidents. To search for false positives, we randomly sampled 10 runs from the top-rated 50 (excluding the three already discussed) and studied them in some detail. All of the 10 exhibited accidents or other abnormal behaviour, suggesting that the false positive rate was low.

True negatives (runs that have no accidents and produce a low score) are perhaps the least significant category. However, the higher the proportion of true negatives, the more situations need to be generated (and runs performed) before problems are identified. In total, 227 runs (45% of the total) were assigned a score of less than ten points. 10% of these were sampled, and in all cases the low ratings were justified. This sampling also suggests that the false negative rate was low.

5 Related Work

As noted earlier, there are many applications of procedural generation in video games and virtual environment training (e.g. [2, 4-7]). These, however, aim for an optimum appearance or a slightly varied play experience, rather than the highly diverse challenges that concern us here. The use of Parish and Müller [15] in the CityEngine

urban planning system is closer to our concerns, but still not directly applicable to creating challenging environments for robots.

Ashlock et al. [16] presented a technique for generating mazes to test the effectiveness of robot path planning algorithms. They evolved mazes which they then statically analysed to check they meet specific constraints, such as a minimum number of turns or a minimum optimal route length. Unfortunately, they did not evaluate any path-planners in the generated mazes – they merely conclude that high fitness scores were achieved and that the visualised output was varied and (intuitively) challenging.

Nguyen et al. [17] apply a similar approach, using evolutionary optimisation to create challenging test situations for an AR. They then take it one step further, putting simulated AR within the situations to test how well they perform. The robot’s performance is measured using a fitness function based on stakeholder-derived “soft goals”, and this is used to guide the optimisation. In this their technique is successful (it produces high fitness values), but they do not explore whether the runs are revealing diverse faults or merely exploiting progressively worse consequences of a single fault.

There are a variety of approaches to runtime safety of AR (e.g. Wardziński [18]). These are orthogonal to the design-time analysis described in this paper; both are necessary for safe AR. Indeed, there would be interesting further work in applying our approach to analysis of a system that included such a runtime safety system, looking for situations that could cause the runtime system to dangerously fail.

6 Conclusions

We have described a PCG approach for generating situations to find faults in robot control software, and shown through a small case study that the basic approach can find faults in some cases. The case study found faults in the standard SND algorithm, without being tuned specifically for application to SND. The method is not fully automated – it requires a human engineer to study accident runs and discover exactly what is causing the problems – but the tool generates situations, executes them, and prioritises the results for human attention.

Because the method tests purely by generating environments, it could be used to test any robot that is written for Player/Stage. Similarly, it can be used on a simple prototype in order to test an algorithm or an overall control strategy – a lot can be learned (fundamental algorithm flaws can be found) without needing a fully developed and mature implementation.

As further work, this approach could be implemented in a higher-fidelity simulation. This is perhaps best done by industry or as a commercialisation effort; further academic work should focus on refining the situation generation and result prioritisation algorithms, and understanding the space of environments that cause problems for major AR algorithms and strategies.

An empirical evaluation of the method’s fault-finding power would be valuable. It could be compared to conventional testing approaches, and to manual situation generation. The comparative testing work by Nguyen et al [19] is useful template – in

particular, note the way that they assess proportion of known faults found rather than relying on arbitrary measures such as run scores or search fitness functions.

As further evaluation, there are many robot algorithms and control strategies that this approach could be applied to. One interesting case would be to apply it to the initial formulation of Velocity Obstacles, as presented by Fiorini and Shiller [20], and check whether it reveals the known flaws in that algorithm (see [21], [22]).

Although the case study showed the run scoring system producing useful results, it is not clear whether it is well-calibrated to the scores that a human expert would assign had they studied the situation in detail. It may be possible to learn or evolve the parameters of the scoring system (event weightings) by having human experts rate a large set of runs, then letting the learner or optimiser derive a scoring system that reproduces those scores for those runs. Such scoring system could be used as a fitness function for an optimisation technique that tuned the parameters of the situation generator. Although trying to maximise the scores achieved might lead to a narrow focus on a few high-impact faults, tuning the generator to minimise the number of *low* scores achieved might greatly increase the performance of the approach without compromising the diversity.

Acknowledgements. The authors would like to thank Ibrahim Habli for his comments on an earlier draft of this paper.

References

1. Fletcher, L., Teller, S., Olson, E., Moore, D., Kuwata, Y., How, J., Leonard, J., Miller, I., Campbell, M., Huttenlocher, D., Nathan, A., Kline, F.-R.: The MIT – Cornell Collision and Why it Happened. *Journal of Field Robotics* 25, 775–807 (2008)
2. Martin, G.A., Hughes, C.E.: A Scenario Generation Framework for Automating Instructional Support in Scenario-based Training. In: *Proceedings of the Spring Simulation Multiconference* (2010)
3. Knight, J.C., Leveson, N.G.: A Large Scale Experiment In *N-Version Programming*. In: *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, Ann Arbor, MI, pp. 135–139 (1985)
4. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* (2011)
5. Togelius, J., Preuss, M., Yannakakis, G.N.: Towards multiobjective procedural map generation. In: *Proceedings of the Workshop on Procedural Content Generation in Games (PGGames 2010)*, Monterey, CA (2010)
6. Roden, T., Parberry, I.: From Artistry to Automation: A Structured Methodology for Procedural Content Creation. In: Rauterberg, M. (ed.) *ICEC 2004*. LNCS, vol. 3166, pp. 151–156. Springer, Heidelberg (2004)
7. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: *Search-Based Procedural Content Generation Applications of Evolutionary Computation* (2010)
8. Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based test generation for multiagent systems. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 1315–1320 (2008)

9. Archer, J.: Indicators for traffic safety assessment and prediction and their application in micro-simulation modelling. PhD thesis, KTH, Stockholm (2005)
10. Gerkey, B.P., Vaughan, R.T., Howard, A.: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In: Proceedings of the Intl. Conf. on Advanced Robotics (ICAR), Coimbra, Portugal, pp. 317–323 (2003)
11. Perlin, K.: An Image Synthesizer. SIGGRAPH Comput. Graph. 19, 287–296 (1985)
12. Hart, P.E.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4, 100–107 (1968)
13. Durham, J., Bullo, F.: Smooth Nearness-Diagram Navigation. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2008), pp. 690–695. IEEE (2008)
14. Arnold, J.: Robot Hazard Analysis using Procedural Content Generation. MEng thesis, University of York (2012)
15. Parish, Y.I.H., Müller, P.: Procedural Modeling of Cities. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, New York, USA, pp. 301–308 (2001)
16. Ashlock, D.A., Manikas, T.W., Ashenayi, K.: Evolving A Diverse Collection of Robot Path Planning Problems. In: IEEE Congress on Evolutionary Computation, Vancouver, BC, Canada (2006)
17. Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M.: Evolutionary testing of autonomous software agents. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009 (2009)
18. Wardziński, A.: The role of situation awareness in assuring safety of autonomous vehicles. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 205–218. Springer, Heidelberg (2006)
19. Nguyen, C.D., Perini, A., Tonella, P.: Constraint-based Evolutionary Testing of Autonomous Distributed. In: Proceedings of the Software Testing Verification and Validation Workshop, Lillehammer, pp. 221–230 (2008)
20. Fiorini, P., Shiller, Z.: Motion Planning in Dynamic Environments Using the Relative Velocity Paradigm. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 560–565 (1993)
21. Large, F., Laugier, C., Shiller, Z.: Navigation among moving obstacles using the NLVO: Principles and applications to intelligent vehicles. Autonomous Robots 19, 159–171 (2005)
22. Fulgenzi, C., Spalanzani, A., Laugier, C.: Dynamic Obstacle Avoidance in uncertain environment combining PVOs and Occupancy Grid. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 1610–1616 (2007)

Fine-Grained Implementation of Fault Tolerance Mechanisms with AOP: To What Extent?

Jimmy Lauret^{1,2}, Jean-Charles Fabre^{1,2}, and H el ene Waeselynck^{1,2}

¹ CNRS, LAAS, 7 Av. Du Colonel Roche, 31400 Toulouse, France

² Univ de Toulouse, INP, LAAS, 31400 Toulouse, France

{Jimmy.Lauret, Jean-Charles.Fabre, Helene.Waeselynck}@laas.fr

Abstract. The benefits of using aspect oriented programming (AOP) for separation of concerns is well-known and has been demonstrated in many works, including for dependable computing. In this paper, we use this composition capability of AOP to develop micro-aspects that can be combined together to realize a given fault tolerance mechanism. The toolbox of micro-aspects can be used to make mechanisms easily configurable and by the way to simplify their update. We show that the composition of micro aspects leads to undesirable side effects of the interactions between them, called interferences. We propose an approach to detect interferences with executable assertions, using an extension of AspectJ called AIRIA that enables control over an aspect chain at a shared join point. We finally draw the lessons learnt and discuss to what extent AOP can be used to develop fault tolerance mechanisms.

1 Introduction

The benefits of using aspect oriented programming (AOP) for separation of concerns is well-known and was demonstrated in many works, including for dependability. Mechanisms like *Time Redundancy* and *Control Flow Checking* have been implemented recently in AOP in the automotive context [1]. Fault tolerance mechanisms (FTM) can be implemented in separate software components, the aspects, and later woven at specific locations of the application code, called join points. In practice several mechanisms can be composed together to fulfill different fault tolerance properties. But composition may also be used to develop a single mechanism. For example, a replication mechanism to tolerate a server crash could be designed as the composition of several individual aspects, each of them realizing a small task: handling request sending to multiple destinations, handling request reception depending on the replica mode of operation, handling checkpointing, handling response return to client. This idea leads us to the notion of *micro-aspects*, which is similar to the notion of micro-protocols developed by R. Schlichting et al. [2-3]. It is appealing to resilient computing [4] since mechanisms become easier to configure and to update. By offering fine-grained, reusable micro-aspects, we intend to simplify the job of the developers. The first question we tackle in this paper is thus:

To what extent can fine grain aspects be defined and composed to implement reconfigurable fault tolerance mechanisms?

But the composition of the micro-aspects to realize a replication protocol may also lead to undesirable side effects, called interferences. In other words, the aspect chain might be incorrect after weaving all micro-aspects into the application code. We need to control the composition and provide means to detect undesirable interferences.

The second question we tackle relates to interactions between aspects and how undesirable interferences between aspects can be detected?

We propose to instrument the aspect chain using executable assertions. The proposed approach relies on the concept of *resolver* provided by an extension of AspectJ [5], called AIRIA [6].

The paper is organized as follows. In section 2, we present the interesting features provided by AIRIA to control the composition of aspects. In section 3, we develop a replication mechanism using a collection of micro-aspects. In section 4, we present our instrumentation approach to detect interferences. In section 5 we discuss the benefits and the drawback of using micro-aspects to develop fault tolerance mechanisms. Section 6 concludes the paper.

2 AspectJ and Variants

AspectJ [5] is our target language in this work. When several aspects are woven at the same join point [7], a given ordering is required. AspectJ offers a *declare precedence* statement for this. If no precedence is declared the compiler may choose an arbitrary ordering. Precedence works at the granularity of aspects, not advices, which may be too coarse-grained. For example, assume an aspect *ACrypt* has two advices for encryption and decryption, and another aspect is used for logging messages. We would like to log clear messages, but it is not possible to declare separate precedence policies for logging/encryption on the one hand, and decryption/logging on the other hand.

Finer-grained resolution of conflicts can be found in AspectJ extensions, like the AIRIA extension [6]. It offers a *Resolver* construct to define precedence policies for advices. A resolver is a kind of around advice (cf. Figure 1) used to control the composition of advices at shared join points, i.e. an advice for composing advices. In Figure 1, the resolver `sender` is applied at each join point where the advices `Alog.logMsg` (message), `ACrypt.encrypt` (message ciphering), `AAuth.auth` (sender authentication), are woven (as specified in the `and` clause, line 3). The list between brackets determines the order of execution for the `proceed` clause (line 4). In the example, `AAuth.auth` is applied first, then `Alog.logMsg` is applied and finally `ACrypt.encrypt` encrypts the message before sending.

More complex policies can be defined by using if-then-else statements to select the appropriate `proceed` clause. Also, multi-level policies are possible with resolvers of resolvers. The AIRIA compiler checks that, whatever the join point, a unique root resolver manages conflicts at this join point. A total execution order must be obtained from the tree of resolvers starting from the root.

```

1 aspect LogEncryptAuth {
2 void resolver sender():
3 and(ALog.logMsg, ACrypt.encrypt, AAuth.auth) {           // pointcut selection
4 [AAuth.auth, ALog.logMsg, ACrypt.encrypt].proceed();}   // order before proceed
5 }

```

Fig. 1. The Resolver construct of AIRIA

We found resolvers convenient not only for mastering the ordering of advices, but also for instrumentation purposes. As undesirable interferences may induce subtle failures, we would like to reveal them by means of assertions. Based on the resolvers scheduling the conflicting advices of our micro-aspects, we can add instrumentation advices to the schedule and precisely control their placement in the execution chain.

3 Fine-Grained Development of FTM Using Micro-Aspects

The objective here is to determine a set of reusable micro-aspects to implement an exemplary replication protocol (PBR, *Primary Backup Replication*). In this simple example, we assume reliable point-to-point communication channels and that the system is synchronous. Some implementation details will be skipped for space limitation and also because they are of no interest to define micro-aspects. The example given can be seen as too simple, even naïve, but it is complex enough to illustrate interferences between micro-aspects during its development.

3.1 Client-Side Micro-Aspects

In our simple example, the client handles two communication channels with the primary (`primary_adr`) and with the backup (`backup_adr`) when the primary fails.

Any request includes a `client_id` and a `request_number`, forming a unique request id. An aspect named `ANumbering` that provides two advices, `insert` and `remove`, manages request numbers. Similarly, an aspect named `AIdentifier` that provides two advices, `insert` and `remove`, manages the client id.

A third aspect is responsible for the management of requests in progress, called `ACacheManager`, providing an advice `addin`: `ACacheManager.addIn` stores the request into a cache together with its corresponding timer value.

When the primary fails, the pending request (stored into the cache) must be resent to the *alive* replica (the former backup that is now the primary). The handling of failed requests is done by another aspect called `ARequestResend`. It provides an advice called `reload`: `ARequestResend.reload` retrieves the request into the cache and reloads it for retry. The pointcut expression for this aspect captures the exception raised when the timer expires. The aspect `ATimer` provides two advices `start` (`time_window`) and `stop`.

The various aspects defined above are mostly weaved at two shared join points, namely `send()` and `receive()`. Each aspect executes an elementary action. Their correct composition implements the expected client behavior (cf. 3.3).

3.2 Server-Side Micro-Aspects

A server replica has two operational modes, primary or backup. We define in this section the micro-aspects needed to implement them. The first action is to initialize the operational mode of the replica. An AspectJ field introduction is used to set up the mode for each replica. A micro-aspect named `AServerMode` is responsible for the initialization of the replica's mode. It offers a unique advice `AServerMode.init`.

Micro-Aspects in Primary Mode. The behavior in primary mode is as follows:

- 1) the client sends a request (`request_number`, `client_id`, `request_body`) to the primary server that receives it (unless the server replica crashes).
- 2) the primary server processes the request, captures the state of the computation at the end of the processing step, sends a checkpoint message (`request_number`, `client_id`, `response_body`, `primary_state`) to the backup that stores it, and, lastly, forwards the response to the client (`request_number`, `client_id`, `response_body`).

The algorithm presented in Figure 2 is simplified. The response to the client is part of the checkpoint message because it is necessary in case of primary crash. Suppose that the primary crashes after sending the checkpoint message and just before sending back the response to the client. When the timer corresponding to this request expires, the client re-sends the request to the new primary (the former backup). In this case, the request cannot be processed twice for conservative reasons (*Only Once Semantics*). To this aim, filtering duplicate requests is a mandatory.

When received, a copy of the request is forwarded to the backup that stores it into a cache. When the primary fails, the failure detector triggers the reconfiguration of the backup as a new primary. The new primary can process pending requests stored in the cache. The above described behavior leads to define the following micro-aspects to implement the primary behavior.

The management of the inter-replica protocol in this example is delegated to a micro-aspects named `ACheckpoint` that provides two advices:

- `ACheckpoint.ForwardRequest` forwards the request to the backup;
- `ACheckpoint.BuildCheckpoint` first captures the state of the replica and then prepares the checkpoint message (`request_number`, `client_id`, `response_body`, `primary_state`) before sending it.

```

1 variable of the protocole:    mode    // set to primary
2
3 receive request
4 if (mode=primary) {
5     forward request to backup
6     remove request number from the request
7     remove client_id
8     request processing           // new message only
9     checkpointing :capture state and send checkpoint to backup
10    insert request number in the response message
11    send response to client
12 }

```

Fig. 2. Pseudo algorithm of the primary

To remove the `client_id` and the `request_id` before processing the request (`request_body`), we use: `AIdentifier.remove` and `ANumbering.remove`.

Micro-Aspects in Backup Mode. The behavior in backup mode can be summarized as follows:

- the primary forwards a request (`request_number`, `client_id`, `request_body`) that is stored by the backup into a cache;
- the backup also receives the checkpoint messages from the primary (`request_number`, `client_id`, `response_body`, `primary_state`). The backup updates its current state with the `primary_state` information and stores the response into the cache (`request_number`, `client_id`, `response_body`).

The (`request_number`, `client_id`) is a unique index for sorting information into the cache that can be seen as a hash table.

The management of both the copy of the request message and the checkpoint message are done by the same aspect `ACheckpoint` previously defined. To handle both messages, we add three more advices in `ACheckpoint`:

- `ACheckPoint.putInCache` stores the request into the cache;
- `ACheckPoint.updateCache` stores the checkpoint into the cache;
- `ACheckPoint.updateState` updates the backup state with the primary state.

The receive statement in the backup mode is thus a join point for multiple advices depending on the type of message.

It is worth noting that, in the current implementation, the primary failure event raised by the failure detector triggers an event handler that changes the mode of the backup replica to primary (with no backup until a new backup is installed).

3.3 Composition of Micro-aspects

Micro-aspects Integration for the Client. The micro-aspects `ANumbering.insert`, `ACacheManager.addIn`, `ATimer.start`, `AIdentifier.insert`, are applied at the same shared join point, i.e. before the send statement. These are before advices to be applied in the following order: `ANumbering.insert` < `ACacheManager.addIn` < `AIdentifier.insert` < `ATimer.start`. The precedence order is determined by the resolver in Figure 3. The receive statement is also a shared join point. Figure 4 shows its resolver.

```

1 aspect sendResolution {
2 void resolver sender ():
3 and (AIdentifier.insert, ACacheManager.addIn, ATimer.start, ANumbering.insert) {
4 [ANumbering.insert, ACacheManager.addIn, AIdentifier.insert, ATimer.start].proceed (); }

```

Fig. 3. Resolution of interactions around the send join point

```

1 aspect receiveResolution {
2 void resolver receive ():
3 and(ANumbering.remove , ATimer.stop){
4 [ANumbering.remove , ATimer.stop].proceed();}

```

Fig. 4. Resolution of interactions around the receive join point

Micro-Aspects Integration for Server Replicas. The behavior of the replica depends on the mode, primary or backup. We then use a “*resolver of resolver*” approach. The root resolver `RCheckMode.getMode` determines the mode and then the management of the appropriate advice chain is delegated to a child resolver. A resolver called `RduplexPrimary.run` manages the list of advices in primary mode. A resolver called `RduplexSecondary.run` manages the list of advices in backup mode.

The `RduplexPrimary.run` resolver applies the advices around the service in the following order: `ACheckPoint.forwardRequest` < `AIdentifier.remove` < `ANumbering.remove` < `service` < `ACheckPoint.buildCheckPoint` .

In backup mode, two series of actions must be made, first i) when the request is received, and ii) when the checkpoint is received. These actions are performed by the `RduplexSecondary.run` resolver:

- it determines first whether the received message is a request or a checkpoint;
- when it is a request the advice `ACheckPoint.putInCache` is invoked;
- when it is a checkpoint the advices `ACheckPoint.updateCache` and `ACheckPoint.updateState` are called.

Last but not least, the filtering of duplicate requests is delegated to the `ACheckDuplicate` micro-aspect that provides a unique advice named `reply`. `ACheckDuplicate.reply` first detects duplicate requests and, in this case, extracts the response, if any, from the cache. The response is returned to the client. When no response is retrieved in the cache, the request needs to be processed.

Table 1. Resolvers of resolvers summary

Level	Name	Triggered advice or resolver
Resolver level 0	<code>RCheckMode.getmode</code>	Resolvers level 1
Resolver level 1	<code>RduplexPrimary.run</code> <code>RduplexSecondary.run</code>	<i>Each resolver triggers the advices defined for each mode</i>

In summary, the final implementation uses two levels of resolvers, the behavior of each mode is handled by level 1 resolvers scheduling advices in each case. Level 0 is the root resolver handling level 1 resolvers depending on the mode of operation of the replica (cf. Table 1). Resolver level 0 (the root resolver) checks the mode and triggers `RduplexPrimary.run` or `RduplexSecondary` resolvers accordingly. Resolvers `RduplexPrimary.run` and `RduplexSecondary` trigger in turn the advices defined for each mode.

Discussion. In this section we have shown how micro-aspects can be defined and integrated together to realize a given replication protocol. This integration leads us to insert many micro-aspects at shared join points. The scheduling is managed by resolvers, even by resolvers of resolvers. The interesting point here is that many these micro-aspects are reusable for different variants of replication protocols (`AIdentifier`, `Anumbering`, `ATimer`, `AResend`, `ACacheManager`), some being more specific (`ACheckpoint`). But, even in this simple replication protocol, subtle interactions can lead to interferences, i.e. errors. In Section 4, we propose an approach to prevent and detect interferences.

4 Interference Detection

An interference is an undesirable interaction between aspects woven at a shared join point. During the integration of several micro-aspects at a given join point, some assumptions regarding micro-aspects interactions are implicit, potentially leading to an interference. These assumptions must be made explicit. These properties are transformed into executable assertions in order to verify the composition at runtime.

4.1 The Interference Problem

Interferences are always defined by considering the sequential execution of aspects woven before, after or around a target instruction in the base code (the shared join point). During this sequential execution, side effects occur due to read/write access to shared data (*data-flow interference*) or due to actions affecting the passing of control to the next advice or to the base code (*control-flow interference*). The authors of [8] consider four cases of interference, two dataflow and two control-flow ones:

- *Change Before (CB)*. Aspect A accesses a variable v of the base code, the value of which was changed by other aspects executed before A . A 's behavior might differ from the one we get if the variable v had kept its original value.
- *Change After (CA)*. Aspect A accesses a variable of the base code, the value of which is later changed by other aspects executed after A . Due to the new value of the variable, A 's behavior may be inadequate, or partly cancelled.
- *Invalidation Before (IB)*. Aspects executed before A bring the system to a state which is no longer a join point for A , preventing thus A from executing.
- *Invalidation After (IA)*. Aspects executed after A bring the system to a state which is no longer a join point for A , hence they remove a join point of A .

In this paper, we stick to these four cases. They are sufficient for illustrating the general characteristics of control and data-flow interferences. It is worth noting that an interaction is not necessarily a problem. It depends on the intended behavior of aspects, i.e. the expected behavior for the user. For example, we may judge that A 's purpose is violated if A can be cancelled (IB interference case). We may then augment A 's specification by an explicit statement that A 's execution is mandatory: it eventually occurs after any arrival at a join point for A . Conversely, for another aspect B , it may be acceptable that previously executed aspects put the system in a state no longer requiring the execution of B .

The important questions to be addressed are the following: How to detect interactions of multiples aspects at a shared join point? How to specify the expected behavior? How to validate composition?

4.2 Proposed Approach

The approach is described in detail in [9], we just summarize its main concepts and practical steps in this section. The aim is to detect interferences in the integration of micro-aspects into an application.

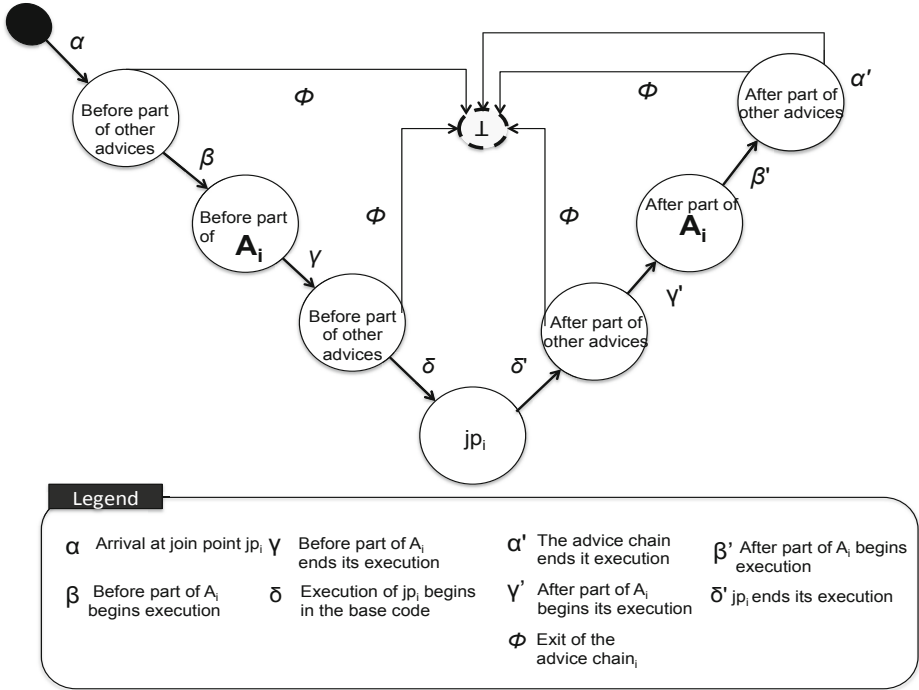


Fig. 5. Lifecycle of an around advice

Overall Principle. First, the designer must declare whether data- or control-flow interactions between a set of aspects are expected or not. Then, the resolution implies defining the correct order of the application of multiple advices at a shared join point. Interferences may induce subtle failures, we propose to reveal them by means of assertions. To keep separation of concerns, the assertions should be implemented with dedicated aspects that monitor the execution of other aspects. The *resolver* construct is used to this aim.

Figure 5 exemplifies the lifecycle of an around advice A_i , where several advices are attached to a join point jp_i . We distinguish A_i from other advices, because it is of interest for a non-interference property (e.g., we want to forbid a CB interference on a data read by A_i). Transition α represents the passing of control from the base code to the first advice, β is the activation of the before part of the distinguished advice, etc. After the execution of the join point, conflicting advices are popped in the reverse order of precedence (δ', \dots, α'). At any time, the control flow may get out of the chain of advices (ϕ transitions). All these transitions must be reified for the instrumentation of the composition. For example, to observe a CB, we need to record a data value at α and detect its change at β . Unlike AspectJ, AIRIA does expose transfers of control between the base level and the aspects, like α and δ , thanks to the *resolver* construct. In [9], we demonstrated that we can precisely control the placement of instrumentation code at any transition of Figure 5. We provided a working solution that was prototyped on artificial examples with randomly generated trees of resolvers. It allows us to automatically instrument the code to detect data- and control-flow interferences.

Detection of Data-Flow Interferences. The detection of both CB and CA consists of the use of two dedicated aspects. `AStorer` provides a monitoring advice that stores the values of selected variables at some point of the aspect chain. `AChecker` checks that the values are unchanged at some later point of the chain. For example, detection of a CB interference affecting the before part of a distinguished before advice A_i involves placing the storer at α and the checker at β . It is straightforward to apply a similar approach to the verification of properties attached to the after part of A_i (e.g., the values at transition γ' should be the same as at δ'), or even to properties spanning the before and after parts (e.g., the values at γ' should be the same as at α).

Detection of Control-Flow Interferences. The detection of IB and IA uses flags to represent the occurrence of expected events in the chain of advices. The initial value of a flag indicates that the event has not happened yet. When the event occurs, the flag value is changed. At the end of the advice chain, the root resolver is able to determine whether an expected event happened or not. For example, IB detection requires us to verify whether A_i is always executed after transition α . A first advice initializes the flag at α and another one sets the flag at β . Pieces of code placed in the root resolver expose a ϕ transition if the flag has not been set. Assume the execution chain is broken (e.g., an advice raises an exception, or does not call `proceed` to pass control to the next advice in the chain): whatever happens, control will come back to the root resolver so that we are able to detect the unset flag. IA detection follows a similar principle.

4.3 Interference Detection in the PBR Implementation

Properties Specification. We present first the expected properties for the correct integration of the previously defined micro-aspects to implement our duplex protocol variant. Our objective is to prevent CB, CA, IB, IA interferences among micro-aspects. We do not detail here the various steps of the interactive specification process but we show some of its result, i.e. the result of the analysis of micro-aspects to be combined at a shared join point.

Let's take an example, i.e. the client-side use of the advice `ANumbering.insert` at a send join point. The duplex protocol integrator must answer questions concerning the weaving of micro-aspects at a send join point.

For instance, a question can be: *“Is there any input variable whose value must not be modified from the join point to the execution of `ANumbering.insert` ?”*. The answer is *yes*: variable `msgContent` from the base code must not be changed before the execution of `ANumbering.insert`. The absence of CB interference must be checked. Such an analysis is applied to every micro-aspect used at a given shared join point.

Integration Properties of Client-Side Micro-Aspects. The properties to be addressed when combining micro-aspects used at the client-side are summarized in Table 2. Column 1 identifies the advice, column 2 the interference, column 2 the data, if any. Row 1 in the table corresponds to the `ANumbering.insert` advice example. A CA detection is attached to the `AIdentifier.insert` advice, Row 2. In our implementation, it is

important to insert first the request number, and then the client id. Row 3 and 4 relate to the advices that store the request in a cache and start a timer. If the request is stored and the timeout later occurs, but the request was actually not sent, we have an IA interference: we want to detect this incorrect behavior.

Table 2. Forbidden interferences for client-side micro-aspects

ANumbering.insert	CB	msgContent
AIdentifier.insert	CA	msgContent
ACacheManager.addin	IA	-
ATimer.start	IA	-

Integration Properties of Primary Mode Micro-Aspects. The properties to be addressed when combining micro-aspects used for the primary are summarized in Table 3.

For the advice AForwardRequest.forward, we must prevent a CB interference through msgContent: no modification of msgContent can be made before the execution of AForwardRequest.forward. In addition, we must verify that the join point is executed after the request has been forwarded to the backup. This is a possible IA interference for AForwardRequest.forward.

For the advice ACheckPoint.buildCheckPoint, we must prevent a CB (respectively CA) interference through msgContent: no modification of msgContent can be made before (respectively after) the execution of ACheckPoint.buildCheckPoint. Finally, we require that the client id is first removed from the message content, and the request number removed last.

Table 3. Forbidden interferences for primary-side micro-aspects

AForwardRequest.forward	CB,IA	msgContent
ACheckPoint.buildCheckPoint	CB, CA	serviceResult
ANumbering.remove	CA	msgContent
AIdentifier.remove	CB	msgContent

Integration Properties of Backup Mode Micro-Aspects. For the backup, the first interference is the same as for the primary. For all other advices, ACheckPoint.{putInCache, updateCache, updateState}, msgContent must not be modified before execution of the advice.

Thanks to this assertion checking approach, four integration faults have been successfully detected during the development of our replication mechanism, two genuine ones and two introduced on purpose.

5 Lessons Learnt

This idea of using micro-aspects like those defined in this paper can be interpreted as pure madness by the reader! In a certain sense it is! Many works in the last 20 years

demonstrated that separation of concerns (with meta-objects, aspects, etc.) was an important paradigm for dependable computing. Since then, aspect-oriented programming moved this idea into practice. Separation of concern is an interesting concept, but non-functional actions in a program can correspond to tiny sets of statements. To what extent this idea is fine for resilient computing? Evolution of software including dependability mechanisms calls for a fine-grain development approach: small pieces can be updated or changed according to the needs. However, the composition of such small pieces may lead to undesirable side effects.

We have demonstrated in this work that aspect oriented programming can be used to develop resilient fault tolerance mechanisms. Using micro-aspects, one can easily understand that developing a variant of a duplex mechanism can be straightforward. An active variant of our passive replication can be done simply by just changing the `ACheckPoint` micro-aspect by a `ASynchronize` micro-aspect. In the active variant, both replicas are active and process input requests, only one replica replies to the client. The `ASynchronize` micro-aspect triggers the execution of the request at the backup as well, the primary sends a notification of request processing completion to the backup, and sends a response to the client (only in primary mode).

However, the use of micro-aspects may be error prone. In particular, it creates many potential sources of interferences between aspects.

Consequently, we have proposed an approach to prevent and detect undesirable interferences. AIRIA's resolver construct allows controlling the order of conflicting advices. It offers finer-grained control than *declare precedence* in AspectJ. Moreover, it forces a total ordering to be defined, hence preventing unspecified cases where the AspectJ compiler chooses an arbitrary order. Also, it offers all the observation points required to instrument a chain of advices. It makes it possible to automatically instrument the code with executable assertions, attaching non-interference requirements to the composition of advices. Undesirable interferences are detected by inserting additional advices to store values, initialize flags, check conditions, etc. We demonstrated the feasibility of the instrumentation for various cases of interferences, exemplifying both data-flow and control-flow effects. This approach was used to validate our micro-aspects based implementation of a duplex protocol.

The resolver construct of AIRIA was very beneficial to solve our problem, but in practice it was complex when using resolvers of resolvers. The instrumentation of an aspect chain is mandatory because the composition is error-prone. We are able to instrument automatically an aspect chain, including handled by resolvers of resolvers [9]. Nevertheless, simpler solutions should be investigated, as programming resolver of resolvers is not an easy task that should be error proof.

Micro-aspect based design and interference detection are the two sides of the same coin! The benefits depend on the capacity to validate the integrated mechanism, i.e. the composition of many micro-aspects. The interest of aspects for dependability is clearly depending on the validation capability we can propose regarding composition. This applied to micro-aspects, at one extreme of the spectrum, but also to the composition of macro-mechanisms. Beyond interferences, point cut definition is a complex issue. For mechanisms like replication, the point cut is often simple (service calls) and should anyway remain simple to convince safety experts.

6 Conclusion

We have done the exercise of using AOP to develop a fault tolerance mechanism trying to take advantage or separation of concerns as much as possible. Using the micro-aspect approach was successful as it enables changing the protocol easily. However, flexibility is not free from a validation viewpoint. The detection of interferences between micro-aspects is a difficult problem. AspectJ does not provide a fine-grain control over aspect composition at a shared join point. This is why we used the resolver construct in the AIRIA extension. From our experience, a hierarchy of resolvers may be complex to use, but can be instrumented for detecting errors.

Our future work will be on the elicitation of conflict resolution policies and associated non-interference requirements. We envision a wizard tool aiding operators to enter scheduling directives and expected properties, before resolver code generation proceeds automatically. We will provide support for this, with an interfacing to the instrumentation solution already existing.

Acknowledgements. This work was partially supported by the IMAP project (Information Management for Avionics Platform) in collaboration with Airbus.

References

1. Alexandersson, R., Öhman, P.: Implementing Fault Tolerance Using Aspect Oriented Programming. In: Bondavalli, A., Brasileiro, F., Rajsbaum, S. (eds.) LADC 2007. LNCS, vol. 4746, pp. 57–74. Springer, Heidelberg (2007)
2. Bhatti, N., Hiltunen, M., Schlichting, R., Chiu, W.: Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Trans. Computer Systems* 16(4), 321–366 (1998)
3. Hiltunen, M., Taiani, F., Schlichting, R.: Reflections on Aspects and Configurable Protocols. In: Filman, R.E. (ed.) 5th International Conference on Aspect-Oriented Software Development, pp. 87–98. ACM Press (2006)
4. Laprie, J.-C.: From Dependability to Resilience. In: Fast Abstracts of DSN 2008, Anchorage (2008), http://www.ece.cmu.edu/~koopman/dsn08/fastabs/dsn08fastabs_laprie.pdf
5. Colyer, M., Clement, A.: Aspect-Oriented Programming with AspectJ. *IBM Systems Journal* 44(2), 301–308 (2005)
6. Takeyama, F., Chiba, S.: An Advice for Advice Composition in AspectJ. In: Baudry, B., Wohlstädter, E. (eds.) SC 2010. LNCS, vol. 6144, pp. 122–137. Springer, Heidelberg (2010)
7. Hilsdale, E., Hugunin, J.: Advice Weaving in AspectJ. In: Murphy, G.C., Lieberherr, K.J. (eds.) 3rd International Conference on Aspect-Oriented Software Development, pp. 26–35. ACM Press (2004)
8. Katz, E., Katz, S.: User Queries for Specification Refinement Treating Shared Aspect Join Points. In: Fiadeiro, J.L., Gnesi, S., Maggiolo-Schettini, A. (eds.) 8th IEEE International Conference on Software Engineering and Formal Methods, pp. 73–82. IEEE Computer Society (2010)
9. Lauret, J., Waeselynck, H., Fabre, J.-C.: Detection of Interferences in Aspect-Oriented Programs Using Executable Assertions. In: ISSRE Workshops 2012, pp. 165–170. IEEE (2012)

Formalisation of an Industrial Approach to Monitoring Critical Data

Yuliya Prokhorova^{1,2}, Elena Troubitsyna², Linas Laibinis²,
Dubravka Ilić³, and Timo Latvala³

¹ TUCS – Turku Centre for Computer Science

² Åbo Akademi University, Joukahaisenkatu 3-5 A, 20520 Turku, Finland

³ Space Systems Finland, Kappelitie 6 B, 02200 Espoo, Finland
{Yuliya.Prokhorova,Elena.Troubitsyna,Linas.Laibinis}@abo.fi,
{Dubravka.Ilic,Timo.Latvala}@ssf.fi

Abstract. A large class of safety-critical control systems contains monitoring subsystems that display certain system parameters to (human) operators. Ensuring that the displayed data are sufficiently fresh and non-corrupted constitutes an important part of safety requirements. However, the monitoring subsystems are typically not a part of a safety kernel and hence often built of SIL1–SIL2 components. In this paper, we formalise a recently implemented industrial approach to architecting dependable monitoring systems, which ensures data freshness and integrity despite unreliability of their components. Moreover, we derive an architectural pattern that allows us to formally reason about data freshness and integrity. The proposed approach is illustrated by an industrial case study.

Keywords: Fault-tolerance, data monitoring systems, formal modelling, Event-B, data freshness, data integrity.

1 Introduction

Data Monitoring Systems (DMSs) are typical for a wide range of safety-critical applications, spanning from nuclear power plant control rooms to individual healthcare devices. Data monitoring is usually not a part of the system safety kernel and hence DMSs are often developed using methods prescribed for SIL1 or SIL2 systems. However, data monitoring might have serious *indirect* safety implications. Indeed, based on the displayed data the operator should take appropriate and timely decisions. Therefore, we have to guarantee that a DMS outputs data that are sufficiently fresh and non-corrupted.

One possible solution would be to build a DMS from highly reliable components and formally verify its correctness. However, such a solution would be rather cost-inefficient. Instead, another practical solution has been recently proposed in the industrial setting¹. The solution is based on building a networked DMS over (potentially unreliable) components and utilising diversity and redundancy to guarantee dependability of a DMS.

¹ We omit a reference to the actual product due to confidentiality reasons.

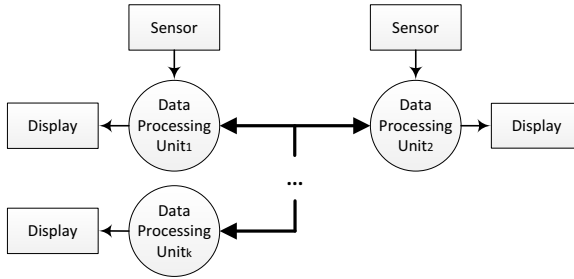


Fig. 1. Distributed monitoring system

In this paper, we aim at giving a formal justification for the proposed industrial solution. We formally define the generic architecture of a networked DMS, formalise the data freshness and integrity properties, and derive the constraints that a DMS should satisfy to guarantee them. We use the Event-B [1] formalism and the associated RODIN platform [2] to formally specify the system architecture and its properties. The proposed specification can be seen as a pattern for designing a networked DMS. We believe that the presented work gives a good demonstration of how formal modelling can facilitate validation of an industrial solution.

2 Industrial Solution to Monitoring Critical Data

In this section, we present a generalised version of the proposed industrial solution to data monitoring. The main purpose of the system is to display a certain system parameter (e.g., temperature, pressure, etc.). We start by defining a generic system architecture.

2.1 Overview of a Distributed DMS Architecture

The monitored parameter is measured by sensors. Each sensor is associated with the corresponding data processing unit (DPU) that periodically reads sensor data. The proposed industrial solution is to build a networked DMS to achieve reliable monitoring of data. The networked DMS contains two types of DPUs – the ones that are directly connected to the sensors and the others that are not. Both types of DPUs output data to the displays connected to them, i.e., the operator observes several versions of data (typically up to four). A generic architecture of the system is shown in Fig. 1.

A network built over DPUs allows them to communicate with each other. The DPUs that are connected to sensors periodically poll sensor data, process data received from sensors and other DPUs, and output the result to the displays as well as broadcast the own processed data over the network. The DPUs that are not connected to sensors perform the same steps, except reading and processing sensor data. The DPUs run different versions of software, i.e., rely on software diversity to avoid common errors. The main goal of the system is to guarantee that each DPU displays only the data that are sufficiently fresh

and non-corrupted. If a DPU cannot satisfy these properties, it should output a special predefined error value.

2.2 Data Freshness and Integrity

Let us now discuss the mechanism of achieving data freshness and integrity. Each DPU has a data pool. In this pool the DPU records the processed sensor data (if the DPU is connected to a sensor) as well as the data received from the other units. The DPU puts in the pool only the data that *have been checked to be non-erroneous*. For sensor data, this means that the obtained sensor reading has passed the reasonableness check and the sensor data processing has completed successfully, i.e., no failure flag was raised. For the data received from the other units, the check of their attached checksums has to be successful and the received data packet should not contain an error message.

Each data processing unit has its own local clock. The system periodically sends a special clock adjusting signal to each DPU to prevent an unbounded local clock drift. All the data that are processed by the system are timestamped. Each unit timestamps every data that it processes based on its local clock. To ensure freshness of the displayed data, before displaying data, the DPU analyses its data pool and filters out the data that are not fresh enough. To select or calculate the data item to be displayed, the DPU applies a predefined function (e.g., maximum) to the set of fresh pool data.

The data are considered to be fresh if the difference between the current (local) DPU time and the data timestamp is less than δ time units. Globally, the freshness property can be formulated as follows: the displayed data are considered fresh if their timestamp differs by no more than $\delta + \epsilon$ time units from the imaginary global clock, where ϵ is the upper bound of the local clock drift.

Data freshness and correctness depend on several factors. If the DPU is connected to a sensor, processing sensor data might take excessive time (e.g., due to a software error) and hence the DPU's own data might not be fresh anymore. Due to network delays or slow processing in other DPUs, the received data might be old as well. Moreover, software errors might corrupt the DPU's own data. A received data packet might also get corrupted during transmission. However, despite a potentially large number of various faults, an occurrence of the system failure, making all DPUs to display an error message, is rather unlikely. Our modelling formally defines the link between data freshness and data integrity that allows us to validate this claim.

3 Formal Generic Development of Distributed Monitoring Systems in Event-B

Let us observe that the generic architecture of DMS described in Section 2 is a composition of loosely coupled asynchronous components. Indeed, each DPU has its own display and relies not only on the data received asynchronously from the other DPUs but also on its own data to produce the displayed data. The system is modular and behaviour of its modules, DPUs, follows the same generic

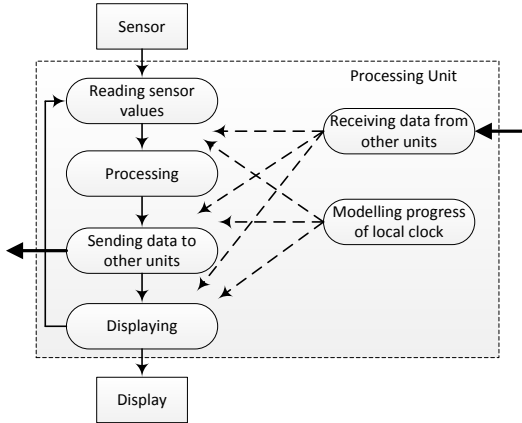


Fig. 2. Dynamics of a DPU

pattern. Therefore, to reason about the overall system, it is sufficient to model the behaviour of its single module and define its interactions with the other modules as a part of the environment specification. One of the obvious benefits of such an approach is clear reduction of the model complexity.

3.1 Abstract Model

Next we will present an Event-B development of a DPU – a generic module of a DMS. Since it is based on the generic architecture of the system discussed above, the presented development is also generic and thus can be instantiated to accommodate for specific details of a concrete monitoring system.

We employ the following refinement strategy. The initial abstract specification formally describes the essential functional behaviour of a DPU. Nevertheless, this allows us to formulate (as model invariants) and verify the desired freshness and correctness properties for the displayed data. The next model (first refinement) introduces fault-tolerance mechanisms and allows us to formulate and prove the required data integrity properties. Finally, the second refinement step deals with the local clock adjustment.

Essentially, the behaviour of a DPU is cyclic. At each cycle, it reads and processes sensor data, broadcasts the processed data to the other DPUs, possibly receives data from them, and finally produces the value to display. These activities are modelled by the events *Environment*, *Processing*, *Sending_Packet*, and *Displaying*. The event *Receiving_Packets* models interaction with the environment – asynchronous receiving of data packets from the other DPUs. The event *Time_Progress* models progress of the local clock. The dynamic DPU behaviour is graphically presented in Fig. 2. The solid lines show the passage of control between the cyclically executed events. Enabledness of asynchronous events is depicted with the dashed lines. The overall structure of the initial specification, the **machine** DPU, is shown in Fig. 3, while Fig. 4 presents its main events.

```

machine DPU
variables main_phase, monitored_value, processed_value, timestamp, displayed_value, curr_time,
           time_progressed, packet_sent_flag

invariants
  main_phase ∈ MAIN_PHASES // phases of the unit cyclic behaviour
  monitored_value ∈ ℕ // raw sensor readings
  processed_value ∈ 0 .. UNIT_NUM → MIN_VAL .. MAX_VAL // collected processed data from all DPUs
  timestamp ∈ 0 .. UNIT_NUM → ℕ // collected timestamp values from all DPUs
  displayed_value ∈ ℕ // the output data
  curr_time ∈ ℕ // the current value of the local unit clock
  time_progressed ∈ BOOL // the flag to determine time progress
  packet_sent_flag ∈ BOOL // the flag to determine sending of a packet
  // Freshness1 ∧ Freshness2 ∧ Correctness

events
  INITIALISATION // initialising variables
  Environment // reading sensor values
  Processing // processing sensor data
  Sending_Packet // broadcasting data packet to other DPUs
  Displaying // outputting data to a display
  Receiving_Packets // receiving packets from other DPUs
  Time_Progress // modelling progress of local clock

end

```

Fig. 3. Outline of the abstract specification

In the model, the variable *main_phase* stores the current phase of DPU execution. The type of *main_phase* is defined as the enumerated set *MAIN_PHASES* of elements $\{ENV, PROC, DISP\}$. Here, the *ENV* phase stands for environment (sensor readings), *PROC* – for data processing, and *DISP* – for data displaying. Broadcasting data to the other units is modelled as a part of the *DISP* phase.

The *Environment* event models sensor reading. As a result, it updates the variable *monitored_value*. As a part of the environment action, we also model a possible adjustment (synchronisation) of the local clock, the value of which is stored in the variable *curr_time*.

The *Processing* event specifies a conversion of the sensor data. We use the abstract function *Convert* to model generic conversion process. The result of the conversion is then used to update the DPU data pool. Implicitly, the event also models a possibility of conversion failure. In this case, the corresponding data pool value remains unchanged (i.e., the last good value is used instead).

To avoid unnecessary complex data structures, we represent the DPU's data pool by two array variables – *processed_value* and *timestamp*. For each $i \in 0..UNIT_NUM$, the data item *processed_value*(*i*) contains the data produced or received from the DPU_{*i*}, while *timestamp*(*i*) contains the corresponding data timestamp. Here the abstract constant *UNIT_NUM* stands for the maximal index value of these arrays (i.e., the number of the DPUs of the system). The value of *UNIT_NUM* can vary for different DMSs. Another generic constants, *MIN_VAL* and *MAX_VAL*, specify the minimal and maximal valid values for the processed measurements respectively.

The *Sending_Packet* event models broadcasting the processed DPU data as data packets to the other DPUs. Each DPU cycle finishes with the execution of the *Displaying* event that calculates the value to be displayed. First it filters (using the relational image operator [...]) the data pool for fresh data and then

<pre> event Receiving_Packets any p where p ∈ PACKET packet_time(p) > timestamp(packet_unit_id(p)) packet_data(p) ∈ MIN_VAL .. MAX_VAL main_phase ≠ ENV time_progressed = TRUE then time_progressed := FALSE timestamp(packet_unit_id(p)) := packet_time(p) processed_value(packet_unit_id(p)) := packet_data(p) end event Sending_Packet any p where main_phase = DISP time_progressed = TRUE packet_sent_flag = FALSE p ∈ PACKET packet_unit_id(p) = 0 packet_time(p) = curr_time packet_data(p) = Convert(monitored_value) then time_progressed := FALSE packet_sent_flag := TRUE end </pre>	<pre> event Processing where main_phase = PROC time_progressed = TRUE then main_phase := DISP time_progressed := FALSE timestamp, processed_value : timestamp' ∈ 0 .. UNIT_NUM → ℕ ∧ processed_value' ∈ 0 .. UNIT_NUM → MIN_VAL .. MAX_VAL ∧ ((timestamp'(0) = curr_time ∧ processed_value'(0) = Convert(monitored_value)) ∨ (timestamp'(0) = timestamp(0) ∧ processed_value'(0) = processed_value(0))) end event Displaying any ss, DATA_SET where main_phase = DISP time_progressed = TRUE packet_sent_flag = TRUE DATA_SET ⊆ ℕ ss = {x↦y ∃i · i ∈ dom(timestamp) ∧ x = timestamp(i) ∧ y = processed_value(i)} [curr_time - Fresh_Delta .. curr_time] (ss ≠ ∅ ⇒ DATA_SET = ss) (ss = ∅ ⇒ DATA_SET = {ERR_VAL}) then main_phase := ENV time_progressed := FALSE packet_sent_flag := FALSE displayed_value := Output_Fun(DATA_SET) end </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 4. Events of the abstract model

applies the abstract function *Output_Fun* on the filtered data to produce the DPU output value to be displayed. If there are no fresh data in the pool, a pre-defined error value (modelled by the abstract constant *ERR_VAL*) is displayed.

Obviously, to reason about data freshness, we should model progress of time. The event *Time_Progress* forcefully alternates between any cyclic events of the model and non-deterministically increases the value of the variable *curr_time*. Event alternation is enforced by using the boolean variable *time_progressed*.

Finally, let us discuss communication between DPUs. It is organised via sending and receiving packets of data. At this level of abstraction, we assume that each packet includes the following fields: (1) an id (i.e., the identification number) of DPU that sent the packet; (2) a timestamp, indicating when the packet was sent; (3) the actual data. We further elaborate of the packet structure, i.e., extend it with new fields, at the next refinement steps.

To access the packet fields, we introduce the following abstract functions:

$$\begin{aligned}
 & packet_unit_id \in PACKET \rightarrow 0 .. UNIT_NUM, \\
 & packet_time \in PACKET \rightarrow \mathbb{N}, \\
 & packet_data \in PACKET \rightarrow MIN_VAL .. MAX_VAL.
 \end{aligned}$$

They allow us to extract the corresponding packet fields. The incoming packets are modelled as parameters of the event *Receiving_Packets*. The extractor functions are then used to decompose these packets. As a result of the event, the pool values *processed_value(j)* and *timestamp(j)* may get updated, where *j* is the index of the DPU that sent the data. However, the update occurs only if the received data are fresher than the previously stored values and the packet did not contain an error flag.

Note that the outgoing packets are constructed (using the same functions) as the local variables of the event *Sending_Packet*.

Our modelling allows us to formally define and verify the data freshness and integrity properties. We define them as model invariants as follows:

Freshness 1: $main_phase = ENV \wedge displayed_value = ERR_VAL \Rightarrow$
 $(\forall i \cdot i \in dom(timestamp) \Rightarrow$
 $timestamp(i) \notin curr_time - Fresh_Delta .. curr_time)$

Freshness 2: $main_phase = ENV \wedge displayed_value \neq ERR_VAL \Rightarrow$
 $\neg\{x \mid \exists j \cdot j \in dom(timestamp) \wedge x = timestamp(j)\} \cap$
 $curr_time - Fresh_Delta .. curr_time = \emptyset$

Correctness: $main_phase = ENV \wedge displayed_value \neq ERR_VAL \Rightarrow$
 $displayed_value = Output_Fun(\{x \mapsto y \mid$
 $\exists i \cdot i \in dom(timestamp) \wedge x = timestamp(i) \wedge$
 $y = processed_value(i)\}[curr_time - Fresh_Delta .. curr_time])$

where *dom* and [...] are respectively the relational domain and image operators, while *Fresh_Delta* is the pre-defined constant standing for the maximum time offset while the data is still considered to be fresh.

The first invariant states that the unit displays the pre-defined error value only when there are no fresh data produced by at least one unit. The second invariant formulates the opposite case, i.e., it requires that, if some data other than the pre-defined error value are displayed, they are based on the fresh data from at least one unit. The third invariant formulates the correctness of the displayed data – the data are always calculated by applying the pre-defined function (i.e., *Output_Fun*) to the filtered fresh data from the unit data pool. The invariant properties are proved as a part of the model verification process.

3.2 Model Refinements

The First Refinement. The aim of our first refinement step is to introduce modelling of failures. We explicitly specify the effect of three types of failures: sensor failures, sensor data processing failures, and communication errors. If the DPU experiences sensor or sensor data processing failures, it does not update the value of its own data in the data pool. Similarly, if the DPU detects a communication error, it does not update the data of the sending process in the data pool. We also abstractly model the presence of software faults, although do not introduce explicit mechanisms for diagnosing them. In all these cases, the mechanism for tolerating the faults is the same: the DPU neglects erroneous

or corrupted data and relies on the last good values from the DPUs stored in the data pool to calculate the displayed value (provided it is still fresh at the moment of displaying).

To implement these mechanisms, first we extend the data packet structure with two new fields: the one containing the information about the status of the DPU that sent the packet, and the other one storing a checksum for determining whether the packet was corrupted during the transmission:

$$\begin{aligned} packet_status &\in PACKET \rightarrow STATUS, \\ packet_checksum &\in PACKET \rightarrow \mathbb{N}, \end{aligned}$$

where the set $STATUS$ consists of two subsets NO_FLT and FLT modelling the absence or presence of faults of a unit respectively. To calculate a checksum, we define the function

$$Checksum \in \mathbb{N} \times MIN_VAL..MAX_VAL \rightarrow \mathbb{N}.$$

The function inputs are the transmitted timestamp and measurement data.

The communication between units is modelled by the event *Receiving_Packets* (see Fig. 5). The event models a successful reception of packets, i.e., when the sending DPU has succeeded in producing fresh data and the corresponding packet was not corrupted during the transmission. If it is not the case, the data pool of the receiving DPU is not updated, i.e., this behaviour corresponds to *skip*.

The detection of sensor faults is modelled by the new event *Pre_Processing*. An excerpt from the specification of this event, shown in Fig. 5, illustrates detection of the sensor fault “*Value is out of range*”. The event *Pre_Processing* also introduces an implicit modelling of the effect of software faults by non-deterministic update of the variable *unit_status*.

At this refinement step, we split the abstract event *Processing* into two events: *Processing_OK* and *Processing_NOK*. The event *Processing_OK* models an update of the DPU’s data pool with the new processed measurements, i.e., it is executed when no failure occurred. Correspondingly, the event *Processing_NOK* is executed when errors have been detected. In this case, the DPU relies on the last good value in its further computations.

The performed refinement step allows us to formulate the data integrity property as the following model invariants:

$$\begin{aligned} \textbf{Integrity 1: } &\forall j \cdot j \in 0..UNIT_NUM \Rightarrow \\ &Checksum(timestamp(j) \mapsto processed_value(j)) = checksum(j) \end{aligned}$$

$$\textbf{Integrity 2: } \forall j \cdot j \in 0..UNIT_NUM \Rightarrow status(j) \in NO_FLT$$

These invariants guarantee that the displayed data are based only on the valid data stored in the DPU data pool. In other words, neither corrupted nor faulty data are taken into account to compute the data to be displayed.

The Second Refinement. The aim of our last refinement step is to refine the mechanism of local clock adjustment. Every k cycles, the DPU receives the reference time signal and adjusts its local clock according to it. This prevents an unbounded local clock drift and allows the overall system guarantee “global” data freshness as discussed in Section 2. For brevity, we omit showing the details of this specification. The complete development can be found in [3].

<pre> event Pre_Processing where main_phase = PROC time_progressed = TRUE pre_proc_flag = TRUE then pre_proc_flag := FALSE sensor_fault : sensor_fault' ∈ BOOL ∧ ((monitored_value ≥ Sens_Lower_Threshold ∧ monitored_value ≤ Sens_Upper_Threshold) ⇒ sensor_fault' = FALSE) ∧ (¬(monitored_value ≥ Sens_Lower_Threshold ∧ monitored_value ≤ Sens_Upper_Threshold) ⇒ sensor_fault' = TRUE) unit_status :∈ STATUS end event Processing_OK refines Processing where // other guards as in the abstract event pre_proc_flag = FALSE sensor_fault = FALSE ∧ unit_status ∈ NO_FLT then // other actions as in the abstract event pre_proc_flag := TRUE processed_value(0) := Convert(monitored_value) checksum(0) := Checksum(curr_time ↦ Convert(monitored_value)) status : status' ∈ 0 .. UNIT_NUM → STATUS ∧ (∃x · x ∈ NO_FLT ∧ status' = status ◀- {0 ↦ x}) end </pre>	<pre> event Receiving_Packets refines Receiving_Packets any p where // other guards as in the abstract event p ∈ PACKET packet_status(p) ∈ NO_FLT Checksum(packet_time(p) ↦ packet_data(p)) = packet_checksum(p) then // other actions as in the abstract event status(packet_unit_id(p)) := packet_status(p) checksum(packet_unit_id(p)) := packet_checksum(p) end event Processing_NOK refines Processing where // other guards as in the abstract event pre_proc_flag = FALSE ¬(sensor_fault = FALSE ∧ unit_status ∈ NO_FLT) then // other actions as in the abstract event pre_proc_flag := TRUE end </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 5. Events of the first refinement model

Discussion of the Development. Let us point out that the proposed approach is also applicable to formal modelling and verification of DPUs that are not connected to a sensor directly (e.g., *Data Processing Unit_k* in Fig. 1). In this case, we can assume that the DPU operates in the presence of a permanent sensor fault and, therefore, only relies on the data received from the other units. The phases related to sensor reading and processing then could be excluded from the model of such a DPU. In general, DPUs might not be necessarily connected to displays but rather send the produced data to other (sub)systems.

In our development we have focused on the logical aspects of data monitoring – the data freshness and integrity properties. Implicitly, we assume that the time-related constraints have been obtained by the corresponding real-time analysis. Such analysis allows us to derive the constraints on how often sensor data should be read, the DPU worst case execution time (WCET), the upper bound of network delay and how often the local clocks should be adjusted. Usually, this analysis is performed when the system is implemented, i.e., with hardware in the loop. In our previous work, we have also experimented with the verification of real-time properties in Event-B [4] and demonstrated how to assess interdependencies between timing constraints at the abstract specification level.

In the next section, we overview the industrial case study and then present the lessons learnt from the development.

4 Validating an Industrial Solution

4.1 Overview of the Industrial Case Study

Our development presented in Section 3 generalises the architecture of a Temperature Monitoring System (TMS). The TMS is a part of the data monitoring system typical for nuclear power plants. The TMS consists of three DPUs connected to the operator displays in the control room.

The TMS is an instantiation of the generic architecture described in Section 2 and formally modelled in Section 3. DPUs of the TMS monitor readings of the temperature sensors installed in a certain module of the plant.

The system is redundant with the architecture 1oo3 (one out of three). The actual temperature signals are generated by two temperature sensors – Resistance Temperature Detectors [5]. The temperature signal from the first sensor is transmitted to two different DPUs – Unit A and Unit B. The temperature signal from the second sensor is transmitted to the third DPU – Unit C.

After obtaining a temperature signal from the sensor, the DPU processes it and sends the temperature data further to the other DPUs. Then, the trusted temperature is communicated to the operator displays. Usually, each DPU gets all three temperature values. The temperature to be shown to the operator is then chosen as the maximum valid value obtained. If no valid data is available, then the error message is shown to the operator warning him about a TMS error.

To guarantee that the trusted temperature data is shown to the operator, the system has to ensure integrity of the temperature data as well as its freshness. To verify the described system, we instantiate the generic models as follows:

- **Variables:** *monitored_value* becomes *temp_sensor_value*, *processed_value* – *temperature*, and *displayed_value* – *output*. The rest of variables may remain unchanged, since they are not application-specific.
- **Constants:** all constants are assigned the values specific for the TMS.
- **Functions:** we instantiate *Output_Fun* with the function *max*: the temperature to be displayed should be the highest among the valid measurements. Moreover, the function *Convert* can be defined precisely, i.e., the actual physical law can be provided to convert a raw reading into the temperature.
- **Invariants:** we instantiate the invariants that guarantee the preservation of data freshness and data integrity as follows:

Freshness 1: $main_phase = ENV \wedge output = ERR_VAL \Rightarrow$
 $(\forall i \cdot i \in dom(timestamp) \Rightarrow$
 $timestamp(i) \notin curr_time - Fresh_Delta.. curr_time)$

Freshness 2: $main_phase = ENV \wedge output \neq ERR_VAL \Rightarrow$
 $\neg\{x \mid \exists j \cdot j \in dom(timestamp) \wedge x = timestamp(j) \cap$
 $curr_time - Fresh_Delta.. curr_time = \emptyset$

Integrity 1: $\forall j \cdot j \in 0.. UNIT_NUM \Rightarrow$
 $Checksum(timestamp(j)) \mapsto temperature(j) = checksum(j)$

Integrity 2: $\forall j \cdot j \in 0.. UNIT_NUM \Rightarrow status(j) \in NO_FLT$

4.2 Lessons Learnt

Next we discuss our experience in applying the proposed generic development pattern to an industrial case study and describe the constraints that should be satisfied by the implementation, to guarantee dependability of data monitoring.

Instantiating the Generic Development. Since the formal development proposed in Section 3 is generic, to model and verify a TMS, we merely had to instantiate it. This simplified the overall modelling task and reduced it to renaming the involved variables and providing correct instances for generic constants and functions, while afterwards getting the proved essential properties practically for free, i.e., without any additional proof effort. This illustrates the usefulness of involved genericity, where the used abstract data structures (constants and functions) become the parameters of the whole formal development.

In our case, this allowed us to model and verify a distributed system with an arbitrary number of units and sensors. Moreover, the introduced constants became the system parameters that may vary from one application to another. For instance, different sensors may have different valid thresholds, while the error value to be displayed may also depend on a particular type of a display. Furthermore, the software functions used to convert the temperature or calculate the output value may differ even within the same system. Nonetheless, the derived formal proofs of the data freshness and integrity properties hold for any valid values of the generic parameters. We believe that the presented approach can also be used in other domains without major modifications.

Validating an Architectural Solution. The generic development approach has allowed us not only to formally define two main properties of monitoring systems – data freshness and integrity but also gain a better insight on the constraints that the proposed architecture should satisfy to guarantee dependability.

Compositionality and elasticity. Since DPUs should not produce one common reading, a DMS can be designed by composing independent DPUs, which significantly simplifies system design. Such an architecture enables an independent development and verification of each DPU. It also facilitates reasoning about the overall system behaviour, since interactions between the components can be verified at the interface level. Finally, the proposed solution allows the system to achieve elasticity – since each DPU has a pool of data, it can seamlessly adapt to various situations (errors, delays) without requiring system-level reconfiguration.

Diversity and fault tolerance. The system has several layers of fault tolerance – at operator, system, and unit levels. Since the operator obtains several variants of data, (s)he can detect anomalies and initiate manual error recovery. At the system level, the system exceeds its fault tolerance limit only if all N modules fail at once. Finally, at the DPU level, even if all DPUs fail to produce fresh data, a DPU keeps displaying data based on the last good value until it remains fresh. At the same time, software diversity significantly contributes to achieving data integrity – it diminishes the possibility of a common processing error.

Constraints. Our formal analysis has allowed us to uncover a number of the constraints that should be satisfied to guarantee dependability. Firstly, let us observe that if a DPU keeps receiving data packets with corrupted or old data

then after time δ it will start to rely only on its own data. Thus, potentially the system architecture can reduce itself to a single module. To avoid this, the designers should guarantee that the WCET of each module is sufficiently short for the processed data to be considered fresh by the other DPUs. Moreover, we should ensure that the network delays are sufficiently short and the data do not become outdated while being transmitted. Furthermore, it should be verified that the successful transmission rate is high enough for a sufficient number of packets to reach their destinations non-corrupted. Finally, to guarantee “global” freshness, we have to ensure that the local clock drift is kept within the limit, i.e., does not allow DPUs to display old data.

5 Related Work and Conclusions

Traditionally, the problem of data integrity is one of the main concerns in the security domain, while data freshness is much sought after in the replicated databases. However, for our work, a more relevant is the research that focuses on achieving data integrity “from input to output”, i.e., ensuring that a system does not inject faults in the data flow.

Hoang et al. [6] propose a set of Event-B design patterns including a pattern for asynchronous message communication between a sender and a receiver. Each message is assigned a sequence number that is checked by a receiver. Though Hoang et al. rely on the similar technique (timestamps), the goal of their modelling – ensuring correct order of packet receiving – is different from ours. The packet ordering problem was insignificant for our study, because a DPU always checks freshness of received data irrespectively of the order packets are received.

Umezawa and Shimizu [7] explore the benefits of hybrid verification for ensuring data integrity. They focus on finding techniques that would be most suitable for verifying error detection, soundness of system internal states and output data integrity. This work can be seen as complementary to ours – it identifies the techniques that can be used to verify freshness and integrity properties.

In this paper, we have generalised an industrial architectural solution to data monitoring and proposed a formal generic model of a data monitoring system. We formally defined and verified (by proofs) the data freshness and integrity properties. We applied the generic development pattern to verify an industrial implementation of a temperature monitoring system. The proposed generic development pattern can be easily instantiated to verify data monitoring systems from different domains. As a result of our modelling, we received formally grounded assurance of dependability of the proposed industrial solution.

As a future work, it would be interesting to experiment with quantitative system verification, e.g., in order to optimise the performance-reliability ratio.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
2. Event-B and the Rodin platform (2013), <http://www.event-b.org/>

3. Prokhorova, Y., Troubitsyna, E., Laibinis, L., Ilic, D., Latvala, T.: Formalisation of an Industrial Approach to Monitoring Critical Data. TUCS Technical Report 1070 (2013)
4. Iliasov, A., Laibinis, L., Troubitsyna, E., Romanovsky, A., Latvala, T.: Augmenting Event-B Modelling with Real-Time Verification. In: Proc. of Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (2012)
5. Hashemian, H.M.: Measurement of Dynamic Temperatures and Pressures in Nuclear Power Plants. University of Western Ontario – Electronic Thesis and Dissertation Repository (2011), <http://ir.lib.uwo.ca/etd/189>
6. Hoang, T.S., Furst, A., Abrial, J.-R.: Event-B Patterns and Their Tool Support. *Software and Systems Modeling*, 1–16 (2011)
7. Umezawa, Y., Shimizu, T.: A Formal Verification Methodology for Checking Data Integrity. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2005), pp. 284–289 (2005)

Protecting Vehicles Against Unauthorised Diagnostics Sessions Using Trusted Third Parties

Pierre Kleberger and Tomas Olovsson

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Gothenburg, Sweden
{pierre.kleberger,tomas.olvsson}@chalmers.se

Abstract. Wireless vehicle diagnostics is expected to provide great improvements to the maintenance of future cars. By using certificates, vehicles can identify diagnostics equipment for a diagnostics session, even over long distances. However, since the diagnostics equipment contains authentication keys used to authenticate such sessions, it is critical that neither the keys nor the equipment is lost. Such a loss can give unauthorised access to any vehicle accepting these keys until the theft is detected and the certificates are revoked. In this paper, we propose a method to protect vehicles against unauthorised diagnostics sessions. A trusted third party is introduced to authorise sessions, thus we do not rely solely on proper identification and authentication of diagnostics equipment. Our approach enables vehicles to verify the validity of diagnostics requests. It is transparent to the diagnostics protocol being used, supports different levels of trust, and can control what commands are permitted during diagnostics sessions.

Keywords: remote diagnostics, connected car, access control, authorisation protocol, trusted third party.

1 Introduction

The introduction of wireless vehicular services is just in its infancy. Many new services will be brought to the vehicle, and remote diagnostics and software downloads are no exceptions. However, compared to other services, these two are fundamentally different. Not only do they require access to the safety-critical in-vehicle network, but they will also deal with sensitive information and be able to update and change vehicles' configurations and software. Thus, to prevent malicious modification of vehicles, diagnostics access must be properly verified. Currently, such access is only verified through authentication, but this is not enough to ensure the safety of the vehicle.

Public-key cryptography with certificates is proposed to be used for authentication in Vehicle-to-Vehicle (V2V) communications, but mechanisms for revoking these certificates are problematic and are still being researched [1]. Since perfect revocation cannot be assumed and certificates and cryptographic keys are

stored in the diagnostics equipment, unauthorised access to diagnostics equipment or their keys, will give access to any vehicle that accepts authentication through these certificates. Thus, when implementing remote diagnostics and software download, vehicles not only need to know which diagnostics equipment they should communicate with, but also what actions this particular equipment is authorised to perform once its identity has been verified.

In our previous work [2], we have identified the problem of impersonation due to lost or stolen diagnostics equipment or keys. Unless stolen equipment or keys are immediately disabled or revoked, they can be used to diagnose and manipulate vehicles without any further approvals, even over long distances. An attacker may, for example, try to manipulate all vehicles in a parking area. The scalability and damage of such manipulations could be great and must be prevented. Even though the loss of diagnostics equipment may be easy to detect and the corresponding keys easily be revoked, copied authentication keys are much harder, if not impossible, to detect since there may be no traces left behind. Such a key may therefore be used for a much longer time before being identified as stolen and revoked. In this paper, we present an approach to protect vehicles against unauthorised diagnostics sessions. A Trusted Third Party (TTP) that governs authorisation policies is introduced, and a protocol to issue authorisation tickets for diagnostics sessions is presented. Thus, authentication alone will no longer be enough to connect to and diagnose a vehicle, but proper authorisation is also required which essentially eliminates the possibility to attack an arbitrary vehicle. We also discuss other benefits from such an approach with respect to different applications and scenarios.

The rest of this paper is outlined as follows. In next section, related research within the area is presented. In Section 3, we present the diagnostics architecture, which threats are considered, and we outline our approach to address unauthorised access. The details of our approach are then presented in Section 4. The paper concludes with a discussion in Section 5, followed by our conclusion in Section 6.

2 Related Work

The problem with offering secure remote vehicular diagnostics and firmware updates have been recognised by many projects and been part of use-cases when addressing Vehicle-to-X (V2X)-communication. For example, in the EVITA project, as well as in the standardisation of the Intelligent Transportation Systems (ITS) platform, both remote diagnostics and remote firmware updates were considered [3,4].

Within the EVITA project, Idrees *et al.* [5] describe an approach to perform firmware updates relying on Hardware Security Modules (HSMs). Authentication keys are exchanged between the software supplier, the diagnostics equipment, the Communications Control Unit (CCU) that connects external communications to the in-vehicle network, and the receiving Electronic Control Unit (ECU). The security of the communication between the diagnostics equipment and the vehicle

is ensured by the secret keys stored within the HSMs, however, diagnostics equipment is assumed to be trusted, and unauthorised access and use to perform remote diagnostics, is not addressed. Only end-to-end protection for firmware updates is ensured.

Other specific protocols for secure software download and firmware updates have also been proposed [6,7,8]. Yet, they only ensure secure distribution of software (i.e., data authenticity, data integrity, data confidentiality, non-repudiation, and data freshness) and do not address remote diagnostics and authorisation issues.

In [9], Nilsson *et al.* perform a security assessment of a wireless infrastructure used for remote diagnostics and software updates over the air. Their wireless infrastructure consists of a back-end system, a communication link, and the vehicle, but external diagnostics equipment is not considered. Instead, all communication is conducted directly between the back-end system and the vehicle.

General approaches for implementing authorisation and access control to services within vehicular networks have also been proposed [10,11,12]. However, these proposals address a different problem of authorisation, that of preventing vehicles from gaining unauthorised access to services offered by the network infrastructure. In contrast, we address the opposite problem, namely to *prevent unauthorised access, not by the vehicles, but to the vehicles and services offered by them.*

Even though standard protocols, such as RADIUS [13] and Kerberos [14], are possible candidates to authorise such access, drawbacks in these protocols make them less suitable in this context. First, RADIUS already has known vulnerabilities [15]. It also suffers from scalability issues since each pair of RADIUS clients and servers must share a secret. Kerberos has other drawbacks. It requires synchronised clocks and the use of both a ticket-granting ticket and a service ticket is superfluous. Also, neither of these protocols easily support the delivery of security policies to the involved parties. We conclude that as adaptations are in any case needed, a protocol that is tailored for this context is better suited.

3 Background

3.1 Terminology

We use the following definitions:

Authentication: the process of establishing the identity of a subject.

Authorisation: the permission a subject can exercise on an object.

Access Control: the process to determine and enforce a subject's permission to an object in accordance to the authorisation.

A typical remote diagnostics session contains the following entities (see Figure 1): the *diagnostics equipment*, the *vehicle*, the *back-end system*, and an *arbitrary network* that connects the diagnostics equipment both to the vehicle and to the back-end system. The back-end system is a system owned by the vehicle manufacturer and holds information about vehicles, such as configuration parameters and versions of software.

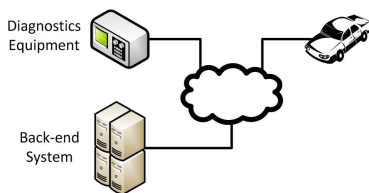


Fig. 1. A typical remote vehicular diagnostics architecture

To perform a diagnostics session, the diagnostics equipment initiates a connection to the Communications Control Unit (CCU) in the vehicle, and if necessary, also to the back-end system to retrieve necessary information about the vehicle. Diagnostics messages are then sent to the CCU, which forwards the messages to the appropriate ECUs in the in-vehicle network.

3.2 Threat Model

The certificate used for authentication (and its associated private key) in the diagnostics equipment is critical to the safety of the vehicle, since the loss of them can lead to impersonation, which can result in unauthorised access to vehicles. The threats against the diagnostics equipment can be of both physical and logical nature [2]:

- **Physical Threats.** An attacker is assumed to be able to *steal* diagnostics equipment, which is able to authenticate itself properly to any vehicle at any time.
- **Logical Threats.** An attacker is assumed to be able to *copy* the authentication keys within the diagnostics equipment. It is then possible to use/have any type of equipment to authenticate as diagnostics equipment to any vehicle at any time.

An attacker is also assumed to be able to perform the following operations on all network traffic between the entities in the diagnostics architecture: *read*, *copy*, *steal*, *modify*, *delete*, *spoof*, *delay* (a combination of steal and spoof), and *replay* (a combination of copy and spoof) [16]. These attacks are threats to the confidentiality and integrity of the transmitted communication, and to the availability of the communicating entities.

3.3 Addressing Unauthorised Diagnostics Access

We address the problem of unauthorised access to vehicles in three steps:

1. One or more Trusted Third Parties (TTPs) that govern security policies for access to vehicles are introduced. The TTPs issue authorisation tickets upon requests from vehicles. The use of tickets is based on the concept of tickets, inspired by the Kerberos system [14].

2. A new authorisation protocol used to request and distribute authorisation tickets to vehicles is proposed.
3. An approach to implement access control in vehicles using the authorisation tickets is presented. Diagnostics equipment is only able to connect to a vehicle after the vehicle has received and verified the issued authorisation ticket.

Before an authorised diagnostics session can be established to a vehicle, a security policy first needs to be defined and stored in a TTP. The diagnostics equipment then initiates the authorisation protocol *before* a “normal” diagnostics session can be established. The result of the authorisation process is that both the vehicle and the diagnostics equipment gets a ticket from the TTP. The ticket contains the cryptographic keys to be used for the diagnostics session and additional information about the session, such as which diagnostics commands are allowed, and for how long access is granted.

4 Authorisation of Sessions

In this section, the details of the proposed approach are given. First, the TTP is introduced. Then, the requirements for the authorisation protocol are given, followed by a detailed description of the protocol, and a security analysis. Finally, an approach to implement fine-grained access control in the vehicle, using the issued authorisation ticket, is given.

4.1 The Trusted Third Party

A Trusted Third Party (TTP) is introduced which governs the security policies that authorises access to vehicles. The purpose of the TTP is to make it impossible to use diagnostics equipment to access vehicles, even if it is able to identify itself using a valid certificate, unless such access also has been granted by an authority. The TTP issues authorisation tickets to vehicles upon request, so that the vehicle can validate the access in accordance to the security policy. The proposed diagnostics architecture, including the TTP, is shown in Figure 2.

The security policies governed by the TTP describe when and for how long a vehicle should grant access to a remote diagnostics session and optionally also what diagnostics commands (messages) are allowed. A security policy is defined by:

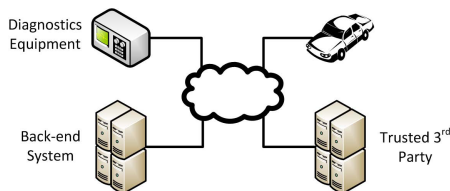


Fig. 2. The proposed authorisation architecture

- a *timespan* for which the authorisation is valid, (t_{begin}, t_{end}) ,
- a set of *authorised equipment*, $\{authorised_equipment\}$, and
- a set of *authorised messages*, $\{diagnostics_message\}$ (optional).

The TTP issues authorisation tickets to the requesting equipment containing a timespan during which it is valid, which can be used by the vehicle to validate the equipment and to filter diagnostics messages in accordance to the set of authorised messages.

The role of a TTP can be undertaken by an organisation external to the automotive company or by a division within the automotive company. Furthermore, the number of TTPs is not limited to only one, but multiple TTPs are possible depending on country, service organisation, and other requirements. The only requirement is that the TTP is acknowledged by the manufacturer, whose vehicles the TTP will serve. The TTP holds a certificate signed by the Certificate Authority (CA) of the manufacturer, which is used to sign authorisation tickets to vehicles and diagnostics equipment. Moreover, all TTPs are assumed to be secure, i.e., unauthorised modifications of security policies are in this context considered to be impossible.

4.2 Protocol Requirements

The following assumptions are made:

- **Certificates.** Each device (see Figure 2) is associated with a certificate and is in possession of the corresponding private key. These certificates have been issued by a CA within the automotive company (or its appointed contractor) and all involved devices trust this CA directly or indirectly as a root of trust. Certificates are not secret and can be distributed without further protection, only the private key must be kept secret. Furthermore, all devices in the architecture are assumed to be in possession of, or be able to retrieve, the certificates needed to perform necessary authentications of other parties. Even if not mentioned explicitly in this paper, we assume that all certificates are validated against certificate revocation lists (CRLs) before being used.
- **Back-end System (BS).** All information needed to diagnose a vehicle is stored in a BS at the automotive company. Examples of such information are configuration parameters and the software available for ECUs in a vehicle. Each automotive company is assumed to have a BS, which can be centralised to one region or distributed over different regions. The BS is assumed to be well protected, i.e., unauthorised changes in the system is not possible.

To guarantee security with respect to the threat model (see Section 3.2), the following security requirements must be fulfilled:

- **Integrity.** An issued authorisation ticket must be protected during transmission against modification, replay, and delay.
- **Confidentiality.** The authorisation tickets must be protected against eavesdropping (read) during transmission. Furthermore, to protect the privacy of vehicle owners, the identity of vehicles should also be protected against eavesdropping during transmission.

Specific approaches to prevent network-based denial-of-service (DoS) attacks against the devices are not considered. The vehicle should ignore traffic from non-authorised equipment and the effect of a network-based DoS attack will only be that new authorisation tickets will be delayed or may not be issued, thereby preventing new diagnostics sessions to be established. However, the vehicle is still protected against unauthorised access.

4.3 The Protocol

The authorisation protocol is shown in Table 1 and the notation used in Table 2. An illustration of the exchanged messages is given in Figure 3. The protocol is divided into two phases. The first phase, message 1–3, identifies the location of the TTP. The second phase, message 4–6, authorises the session between the diagnostics equipment (DE) and the vehicle (V). The security aspects in the protocol will be discussed in next section. After successful completion of step 6, the security policy is known by the vehicle and both the vehicle and the diagnostics equipment will share a common cryptographic session key.

The following messages are exchanged:

1. **Initiate.** DE sends its certificate together with either the certificate of the BS or the TTP to V to request a diagnostics session. We assume that most

Table 1. Authorisation Protocol

(1)	DE → V	$\text{Cert}_{\text{DE}} \parallel (\text{Cert}_{\text{BS}} \mid \text{Cert}_{\text{TTP}})$
(2)	V → BS	$\text{Enc}_{\text{BS}}(\text{V}_{\text{ID}})$
(3)	BS → V	Cert_{TTP}
(4)	V → TTP	$\text{Cert}_{\text{DE}} \parallel \text{Enc}_{\text{TTP}}(\text{V}_{\text{ID}} \parallel \text{nonce}_1)$
(5)	TTP → V	$\text{Enc}_{\text{V}}(\text{Ticket}_{\text{V}} \parallel \text{nonce}_1) \parallel \text{Enc}_{\text{DE}}(\text{Ticket}_{\text{DE}})$ $\text{Ticket}_{\text{V}} = \text{Sign}_{\text{TTP}}(K_{\text{DE},\text{V}} \parallel \Delta t \parallel \text{V}_{\text{policy}})$ $\text{Ticket}_{\text{DE}} = \text{Sign}_{\text{TTP}}(K_{\text{DE},\text{V}} \parallel \Delta t \parallel \text{V}_{\text{ID}})$
(6)	V → DE	$\text{Enc}_{\text{DE}}(\text{Ticket}_{\text{DE}})$

Table 2. Notations

DE	Diagnostics Equipment
V	Vehicle
BS	Back-end System
TTP	Trusted Third Party
Cert_X	Certificate of device X
Sign_X	Signature created by device X, using its private key
Enc_X	Encryption created for device X, using X's public key (also includes a Message Integrity Code)

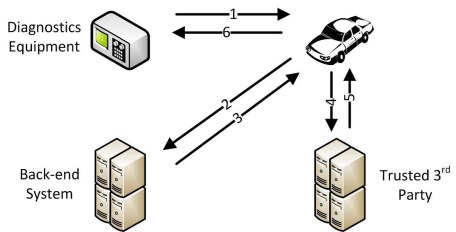


Fig. 3. Messages exchanged during authorisation

DEs already know the identity of the TTP to be used for this session, i.e., a certain TTP is preconfigured in DE. If so, this certificate can be delivered at once to V and step 2–3 can be skipped.

2. **Request TTP Information.** To get the identity of the TTP for this session, V encrypts and sends its identity, V_{ID} , to BS.
3. **TTP Identity.** BS uses the vehicle's ID, V_{ID} , to find the TTP to use for this vehicle and sends its certificate back to V. This completes the first phase of identifying the TTP.
4. **Request Authorisation.** V sends its ID and the certificate of DE to TTP. The time of the request, t_{req} , is recorded by V.
5. **Authorisation Response.** The TTP uses the vehicle's ID, V_{ID} , to find the vehicle's certificate and security policy within its database. A symmetric session key, $K_{DE,V}$, to be used by DE and V is generated and the duration of the authorisation is calculated from the authorised timespan: $\Delta t = t_{end} - t_{current}$. Two authorisation tickets are generated and sent back to V. V_{policy} is optional and may contain what commands DE is allowed to execute.
6. **Protocol Completion.** The security policy and the session key are updated in V and the expiration time of the authorisation is calculated: $t_{exp} = t_{req} + \Delta t$. Since the vehicle knows that $Ticket_V$ is fresh (see next section) and the time of the request, the clocks in V and the TTP do not need to be synchronised. Furthermore, to complete the authorisation, the authorisation ticket is sent encrypted to DE.

4.4 Protocol Security

In this section, we present a security analysis of the authorisation protocol. Each message is analysed with respect to the threats and the security requirements identified earlier in Section 3.2 and 4.2, respectively. To protect confidentiality and integrity of messages, encryption and signatures are used. Furthermore, to prevent replay attacks, nonces are used, i.e., a random number is added to messages belonging to a session so that duplicated, delayed, and replayed messages can be discarded.

The security analysis of each message follows:

1. All certificates have been signed by the CA or an entity trusted by the CA. A certificate chain and the integrity of those are therefore ensured. These certificates can be replaced by a man-in-the-middle (MITM), but since any DE may request an authorisation ticket, the replacement of $Cert_{DE}$ will be regarded as another (but still valid) request. If an authorisation ticket is issued by the TTP, it will be tied to $Cert_{DE}$ and is useless to anyone else not in possession of the certificate's corresponding private key. Furthermore, the replacement of $Cert_{BS}$ or $Cert_{TTP}$ will just result in a failure, since the vehicle's information is not present in other (trusted) BSs or TTPs.
- 2–3. To ensure confidentiality of the vehicle ID, V_{ID} is sent encrypted. Even though the request can be replaced or replayed, it will only tell the location of a TTP for a vehicle to an attacker. However, anyone in possession of

Cert_{BS} can generate such a request to find the TTP associated to a V_{ID} , but this information is not considered to be confidential.

4. As discussed in 1), the certificate is already integrity protected and a replacement of this certificate will only cause an authorisation ticket to be issued for another DE. At first, this may seem to be a problem, however any DE may request an authorisation ticket on behalf of another DE, but the ticket is only useful for the DE it was issued for. Furthermore, to prevent replay attacks and to bind the authorisation request to a certain time, t_{req} is recorded by V and a random number, nonce_1 , is added to the request.
5. To ensure that the issued tickets were generated by the TTP and to prevent modification and eavesdropping of these, they are signed and encrypted. Furthermore, since nonce_1 ties the ticket to a specific session, the freshness of Ticket_V is guaranteed by the request time, t_{req} .
6. This ticket may be replaced or replayed. However, if it is replaced, the session key, $K_{\text{DE},V}$, will not be accepted by V, since the session key was signed by the TTP and is tied to the specific authorisation session identified by nonce_1 in 5). If the ticket is replayed, it will not be accepted by V after the authorisation expires, at t_{exp} .

4.5 Implementing Fine-Grained Access Control

The security policy distributed in each authorisation ticket enables a fine-grained access control mechanism to be implemented in the CCU in the vehicle. The mechanism enforces that only diagnostics equipment for which a valid authorisation ticket has been issued, can send diagnostics messages to the ECUs in the in-vehicle network. It allows the CCU to know what commands this particular DE is allowed to execute. We propose that such a fine-grained access control mechanism is implemented at the network and application layers in the CCU.

To identify authorised diagnostics equipment, the symmetric session key, $K_{\text{DE},V}$, distributed in the authorisation ticket can be used. By adding an HMAC to the payload of each IP packet transmitted between the equipment and the vehicle, mutual authentication can be ensured¹. A simple accept/deny filter can be implemented at the network layer, based on the session key, $K_{\text{DE},V}$, and the expiration time, t_{exp} . Implementing a diagnostics equipment filter at the network layer has major advantages. Diagnostics traffic from non-authorised equipment will be discarded early in the network stack, which limits the exposure of protocols in higher layers in the stack and the possibility to exploit software bugs therein. Such an approach is also independent of, and transparent to, the diagnostics protocol used at the application layer.

To enforce that only authorised diagnostics messages are delivered, a specific diagnostics protocol filter can be implemented at the application layer. Such an approach requires no modification of the diagnostics service, but needs to be adapted to different diagnostics protocols.

¹ The tickets issued by the TTP were encrypted using the public keys of DE and V, thus only the correct DE and V can recover the shared key.

5 Discussion

Without proper authorisation of diagnostics sessions, the loss of diagnostics equipment or its authentication keys can have critical impact on the safety of vehicles. Attackers who acquire diagnostics equipment or its authentication keys can potentially manipulate any vehicle accepting these keys for authentication. By introducing authorisation, such keys cannot be used alone. These keys would only be usable against vehicles scheduled for diagnostics using the authorised equipment. Even though authorisation does not offer complete protection, it prevents large scale attacks and essentially eliminates the possibility to target an arbitrary vehicle.

Our proposed solution has a set of features, which makes it suitable for a large scale implementation and is flexible enough to be used in different scenarios:

- **Independent of the Diagnostics Protocol Used.** The authorisation protocol will be executed before the diagnostics session begins (as stated in Section 3.3). Since communication with diagnostics equipment is validated at the network layer in the CCU, the proposed approach is independent of the diagnostics protocol being used. Hence, the approach is similar to what is done in other security protocols, such as in IPsec. First, authentication is performed and the key material is exchanged (e.g., as in IKE). Then, a protected session can be established that is transparent to the applications (e.g., IPsec communication).
- **Scalable.** The approach is not limited to authorise diagnostics equipment for just one session at a time. By issuing multiple tickets, the same equipment can establish sessions to multiple vehicles at the same time. Hence, the approach is usable for large scale diagnostics sessions and software updates, such as when performing pre-diagnostics of, for example, ten vehicles outside a repair shop, or when performing firmware updates of hundreds of vehicles in a remote harbour before being delivered to car dealers.
- **Different Levels of Trust.** Depending on the diagnostics equipment and the TTP being used, different authorisation tickets can be issued. Hence, different levels of trust are possible. For example, it is possible to have a dedicated TTP for the vehicle inspection authority, which only allows them to read selected information from the vehicle, while another TTP is used by a repair shop which allows their mechanics to both diagnose ECUs and update firmware.
- **Supports Encryption.** The authorisation ticket issued by the TTP holds a symmetric encryption key. This key is not limited to provide mutual authentication and to ensure the integrity of messages, but can also be used to encrypt the whole session between diagnostics equipment and vehicles.
- **No Synchronised Clocks Required.** Since the vehicle records the time of request for an authorisation ticket and the vehicle receives the duration of the authorisation in the ticket from the TTP, no synchronised clocks are needed. This makes it easy to implement in large scale and in a distributed environment.

The number of TTPs are not fixed. This enables the authorisation architecture to be adapted depending on the situation, the organisation, and regulations. An important factor that affects the location of a TTP is the privacy concern of the authorisation information stored. Thus, a TTP within an organisation or even a local TTP within a particular repair shop can be used, so that privacy-related information will not be revealed to other parties.

The approach proposed in this paper is not limited to only diagnostics sessions. The concept of using a TTP to issue authorisation tickets to vehicles is general enough to be used for any service that is going to be provided by vehicles that needs to be authorised. Furthermore, the approach is transparent to application level protocols and is therefore easy to implement.

6 Conclusion

In this paper, we have addressed the problem of lost diagnostics equipment and the loss of authentication keys by proposing a method to authorise diagnostics sessions. Without such a method, an attacker who acquires diagnostics equipment or its private keys can get access to any vehicle accepting these authentication keys.

Three steps are taken to prevent unauthorised diagnostics access. First, a trusted third party, which governs security policies and issues authorisation tickets, is introduced. Secondly, an authorisation protocol to issue authorisation tickets to vehicles is proposed and analysed. Finally, a fine-grained access control mechanism that grant access only to authorised diagnostics sessions is described. The authorisation method ensures mutual authentication and message integrity of diagnostics sessions, filtering of specific diagnostics messages, and supports encryption of diagnostics sessions.

The approach we propose is designed with transparency, adaptation, and minimal changes to current diagnostics protocols in mind. The number of trusted third parties are not limited, which enables them to be placed at different locations and in different organisations as needed. Furthermore, a fine-grained policy can be distributed to the vehicle which allows it to control what service the connecting equipment should get. The filter is transparent to the diagnostics protocol used.

Acknowledgements. This research was funded by the project Security Framework for Vehicle Communication (2011-04434), co-funded by VINNOVA, the Swedish Governmental Agency for Innovation Systems.

References

1. Dressler, F., Kargl, F., Ott, J., Tonguz, O., Wischhof, L.: Research Challenges in Intervehicular Communication: Lessons of the 2010 Dagstuhl Seminar. *IEEE Communications Magazine* 49(5), 158–164 (2011)

2. Kleberger, P., Olovsson, T., Jonsson, E.: An In-Depth Analysis of the Security of the Connected Repair Shop. In: Proc. of the Seventh International Conference on Systems and Networks Communications (ICSNC 2012), November 18-23, pp. 99–107. IARIA, Lisbon (2012)
3. Kelling, E., Friedewald, M., Leimbach, T., Menzel, M., Säger, P., Seudié, H., Weyl, B.: Specification and evaluation of e-security relevant use cases. EVITA Project, Deliverable D2.1, v1.2. (December 30, 2009)
4. ETSI: Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions. Technical Report TR 102 638, v1.1.1, ETSI, 650 Route des Lucioles, F-06921 Sophia Antipolis Cedex, France (June 2009)
5. Idrees, M.S., Schweppe, H., Roudier, Y., Wolf, M., Scheuermann, D., Henniger, O.: Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates. In: Strang, T., Festag, A., Vinel, A., Mehmood, R., Rico Garcia, C., Röckl, M. (eds.) Nets4Trains/Nets4Cars 2011. LNCS, vol. 6596, pp. 224–238. Springer, Heidelberg (2011)
6. Mahmud, S.M., Shanker, S., Hossain, I.: Secure Software Upload in an Intelligent Vehicle via Wireless Communication Links. In: Proc. of the IEEE Intelligent Vehicles Symposium, pp. 588–593 (2005)
7. Hossain, I., Mahmud, S.M.: Secure Multicast Protocol for Remote Software Upload in Intelligent Vehicles. In: Proc. of the 5th Ann. Intel. Vehicle Systems Symp. of National Defense Industries Association (NDIA), Traverse City, MI, National Automotive Center and Vectronics Technology, June 13-16, pp. 145–155 (2005)
8. Nilsson, D.K., Larson, U.E.: Secure Firmware Updates over the Air in Intelligent Vehicles. In: Proc. IEEE International Conference on Communications Workshops (ICC Workshops)., May 19-23, pp. 380–384 (2008)
9. Nilsson, D.K., Larson, U.E., Jonsson, E.: Creating a Secure Infrastructure for Wireless Diagnostics and Software Updates in Vehicles. In: Harrison, M.D., Suján, M.-A. (eds.) SAFECOMP 2008. LNCS, vol. 5219, pp. 207–220. Springer, Heidelberg (2008)
10. Moustafa, H., Bourdon, G., Gourhant, Y.: Providing Authentication and Access Control in Vehicular Network Environment. In: Fischer-Hübner, S., Rannenber, K., Yngström, L., Lindskog, S. (eds.) Security and Privacy in Dynamic Environments. IFIP, vol. 201, pp. 62–73. Springer, Boston (2006)
11. Coronado, E., Cherkaoui, S.: A secure service architecture to support wireless vehicular networks. *International Journal of Autonomous and Adaptive Communications Systems* 3(2), 136–158 (2010)
12. Casola, V., Luna, J., Mazzeo, A., Medina, M., Rak, M., Serna, J.: An interoperability system for authentication and authorisation in VANETs. *International Journal of Autonomous and Adaptive Communications Systems* 3(2), 115–135 (2010)
13. Rigney, C., Willens, S., Rubens, A., Simpson, W.: RFC 2865: Remote Authentication Dial In User Service (RADIUS) (June 2000)
14. Steiner, J., Neuman, C., Schiller, J.: Kerberos: An authentication service for open network systems. In: Usenix Conference Proceedings, vol. 191, p. 202 (1988)
15. Hill, J.: An Analysis of the RADIUS Authentication Protocol (November 24, 2001), <http://www.untruth.org/~josh/security/radius/radius-auth.html>. (verified June 6, 2013)
16. Howard, J.D., Longstaff, T.A.: A Common Language for Computer Security Incidents. (Sandia Report: SAND98-8667) (1998)

Vulnerability Analysis on Smart Cards Using Fault Tree

Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet

Smart Secure Devices (SSD) Team, XLIM/University of Limoges
123 Avenue Albert Thomas, 87060 Limoges, France
{guillaume.bouffard,bhagyalekshmy.narayanan-thampi}@xlim.fr,
jean-louis.lanet@unilim.fr

Abstract. In smart card domain, attacks and countermeasures are advancing at a fast rate. In order to have a generic view of all the attacks, we propose to use a Fault Tree Analysis. This method used in safety analysis helps to understand and implement all the desirable and undesirable events existing in this domain. We apply this method to Java Card vulnerability analysis. We define the properties that must be ensured: integrity and confidentiality of smart card data and code. By modeling the conditions, we discovered new attack paths to get access to the smart card contents. Then we introduce a new security API which is proposed to mitigate the undesirable events defined in the tree models.

Keywords: Smart Card, Security, Fault Tree Analysis, Attacks, Countermeasures.

1 Introduction

A smart card is an intelligent and efficient device which stores data securely and also ensures a secure data exchange. Security issues and risks of attacks are ever increasing and continuous efforts to develop countermeasures against these attacks are sought. This requires clear understanding and analysis of possible ways of attacks and methods to avoid or mitigate them through adequate software and/or hardware countermeasures. To further understand the possible attack paths, a common method is required and in this paper we try to define a method which can help to sort these issues to an extent.

Often countermeasures are dedicated to an attack and do not take into account a global point of view. Usually designers use an inductive (bottom-up) approach where, for each attack a mechanism is added to the system to mitigate the attack. In such an approach, a probable event is assumed and the corresponding effect on the overall system is tried to be ascertained. On one hand, this allows to determine the possible system states from the knowledge of the attacker model and on the other hand the feasible countermeasures for the attack. Thus, it leads to several defenses where the final goal is not clearly defined nor ensured. Moreover the overhead of the defenses is high both in terms of memory footprint and CPU usage.

In this paper, we suggest to adopt a deductive approach (top-down) by reasoning from a more general case to a specific one. We start from an undesirable event and an attempt is made to find out the event, and the sequence of events which can lead to the particular system state. From the knowledge of the effect, we search a combination of the causes. We postulate a state, in our case a security property, and we systematically build the chain of basic events.

This approach is based on a safety analysis, often used for safety critical systems. The safety analysis performed at each stage of the system development is intended to identify all possible hazards with their relevant causes. Traditional safety analysis methods include, *e.g.* Functional Hazard Analysis (FHA) [1], Failure Mode and Effect Analysis (FMEA) [2] and Fault Tree Analysis (FTA). FMEA is a bottom-up method since it starts with the failure of a component or subsystem and then looks at its effect on the overall system. First, it lists all the components comprising a system and their associated failure modes. Then, the effects on other components or subsystems are evaluated and listed along with the consequence on the system for each component's failure modes. FTA, in particular, is a deductive method to analyze system design and robustness. Within this approach we can determine how a system failure can occur. It also allows us to propose countermeasures with a higher coverage or having wider dimension.

This paper is organized as follows, section 3 is about Java Card security. Section 4 describes smart card vulnerability analysis using FTA. Section 5 presents an API to mitigate undesirable events. Conclusions and future works are summarized in section 6.

2 Using FTA for Security Analysis: Related Works

FTA is often used for reliability analysis but it can be also used for computer security. In [3], the authors suggested to integrate FTA to describe intrusions in an IT software and Colored Petri Net (CPN) to specify the design of the system. The FTA diagrams are augmented with nodes that describe trust, temporal and contextual relationships. They are also used to describe intrusions. The models using CPNs for intrusion detection are built using CPN templates. The FTA restricts drastically the possible combination of the events. In [4], FTA is used to assess vulnerability considering the fact that the undesirable events of interest should already be in a fault tree designed for the purpose of a risk assessment. They showed how to build specific FTA diagrams for vulnerability assessments. In [5], an attack tree is used to model potential attacks and threats concerning the security of a given system. Attack trees are always having the same meaning as FTA (same gate connectors). They generate attack scenario in a close way to Unified Modeling Language (UML) scenario for evaluating, for example, a buffer overflow in the system. Another work [6] described a methodology to analyze both software and hardware failure in a system and also discussed the use of FTA affordable for software systems by design, *i.e.* incorporating countermeasures so

that the fault can be considered and mitigated during analysis. Byres *et al.* [7] used the BDMP¹ model to find vulnerabilities in SCADA system.

3 Security of Java Based Smart Cards

Java Card is a kind of smart card that implements the standard Java Card 3.0 [8] specification. Such a smart card embeds a Virtual Machine (VM) which interprets codes already romized with the operating system or downloaded after issuance. Java Cards have shown an improved robustness compared to native applications with respect to many attacks. Java Card is an open platform for smart cards, *i.e.* able to load and execute new applications after issuance. For security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [9]. This protocol ensures that the owner of the code has the necessary credentials to perform the action. Thus different applications from different providers can run on same smart card. Using type verification, the byte codes delivered by the Java compiler and the converter (in charge of delivering compact representation of CLASS files name CAP file) are protected, *i.e.* and the application loaded is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications on the card, enforcing isolation between them. Until now it was safe to presume that the firewall was efficient enough to avoid malicious applications (applet modified after off-card verification). Nevertheless some attacks have been successful in retrieving secret data from the card.

Smart cards are designed to resist numerous attacks using both physical and logical techniques. There are different ways to attack Java Card and many of them are associated with countermeasures. An attacker can use physical or logical attacks [10,11] or combined attacks [12,13] (combination of physical and logical). Often ad-hoc countermeasures can only protect the card from a single type of attacks. So if a manufacturer wants to protect his card against different types of attacks he must embed several countermeasures. Here we are presenting a model to avoid embedding unwanted countermeasures which in turn helps to protect the embedded system against attacks.

4 Smart Card Vulnerability Analysis Using FTA

4.1 Introduction

FTA is an analytical technique (top-down) where an undesirable event is defined and the system is then analyzed to find the combinations of basic events that could lead to the undesirable event. A basic event represents an initial cause which can be a hardware failure, a human error, an environmental condition or any event. The representation of the causal relationship between the events is given through a fault tree. A fault tree is not a model of all possible system failures but a restricted set, related to the property evaluated.

¹ Boolean logic Driven Markov Process.

FTA is a structured diagram constructed using events and logic symbols mainly AND and OR gates (Fig. 1). Other logical connectors can be used like the NOT, XOR, conditional etc. The AND gate describes the operation where all inputs must be present to produce the output event. The OR gate triggers the output if at least one of the input events exists. The INHIBIT gate describes the possibility to forward a fault if a condition is satisfied. For readability of FTA diagrams, a triangle on the side of an event denotes a label, while a triangle used as an input of a gate denotes a sub tree defined somewhere else (Fig. 5).

In our work, we define four undesirable events that represent the failure of the smart card system: code or data integrity, code or data confidentiality. Code integrity is the most sensible property because it allows the attacker to execute or modify the executable code which not only leads to the code and data confidentiality but also the data integrity. The Java Virtual Machine (JVM) prevents these events to occur under normal execution. Intermediate nodes in the tree represent a step in an attack while leafs (basic events) represent either the state of the system or a known attack. If this attack needs a given state of the system to succeed then an AND gate must bind the attack to the state of the system. Thus it becomes a condition on the system state. For example, the EMAN 2 [13] attack shown in Fig. 2, modifies the return address. However, it is necessary to get access to the content of the return address which is represented by the intermediate node on the top. For modifying the return address, the attacker must modify the CAP file and thereby the VM allows the execution of ill typed applet. This latter condition is true if all the basic events are true: absence of an embedded Byte Code Verifier (BCV), no run time check of the index of the local variables and absence of integrity check of the system context area in the frame. The BCV ensures that the applet to be installed is compliance with the Java Card security rules. The presence of one of these countermeasures is enough to mitigate the attack.

4.2 Code Integrity

The first property to be analyzed in a smart card for understanding or implementing security features is the code integrity. If the attacker is able to modify the method area or more generally to divert the control flow of the code to be executed, then he will have the possibility to read or modify the data and code stored inside the card. We consider two possibilities here, see Fig. 3, either the processor executes a new code (execution of an arbitrary shell code) or it executes the regular code in such a way that the initial semantics is not preserved. In the left part of the tree (execution of a shell code), we can obtain it either by changing the control flow by modifying of the branch condition [13] or the return address as explained in the previous section. In the right branch of the fault tree diagram, we modify the execution flow just like a program counter execute an existing code. These attacks refer to code mutation were operands can be executed instead of instructions as explained in [14]. In this class of attack, we can find the attacks mentioned in [15] concerning the Java exception mechanism, the attacks are not only related to the modification of the jump offset but also

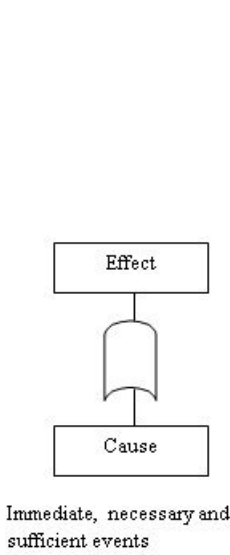


Fig. 1. The causal relationship between events

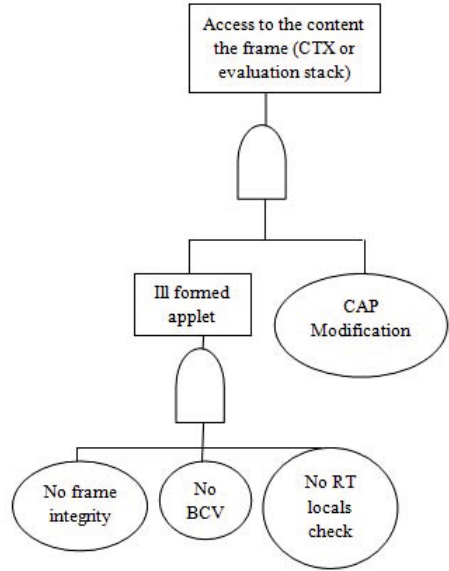


Fig. 2. Steps for an EMAN 2 attack

through a Java Program Counter (jpc) corruption. Then all these intermediate nodes need to be refined to determine all the causes of the unwanted events.

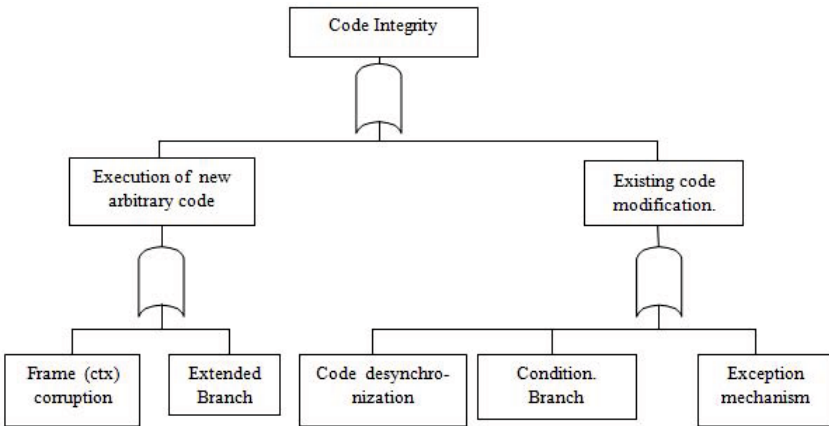


Fig. 3. Code integrity tree

At that step we need to find the most efficient countermeasure to protect the code integrity. As described previously, each attack can be mitigated through a dedicated countermeasures, e.g. for the EMAN 2 attack, it can be either through

the return address stored in a dedicated place (not accessible through a local variable), a frame integrity mechanism, an embedded verifier, verification of the indices of the locals during the load time or some run time checks. One can remark that if the countermeasure is closed to the root of the tree its efficiency in terms of coverage will be higher which doesn't mean that the countermeasure must be close to root. There are other parameters to be taken into account like the run time cost, the latency, the memory footprint etc. For example the Dynamic Syntax Interpretation (DSI) has a high coverage as shown in Fig. 4: this countermeasure is at the top of the tree. A lot of other countermeasures can also be applied here. This work has been extended to three other undesirable events.

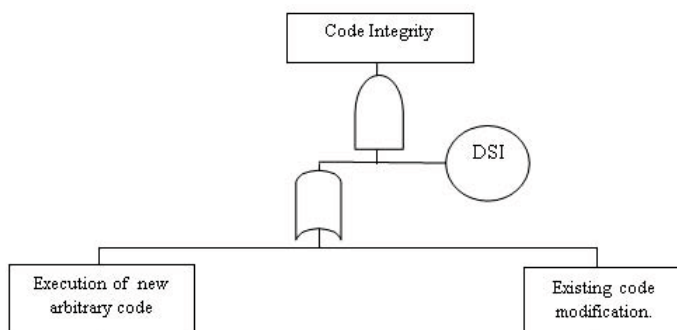


Fig. 4. An efficient countermeasure for code integrity

4.3 Basic Events: Possible Attacks

Known attacks are considered as basic events. Several authors proposed and tested different types of attacks which are discussed here.

Abusing shareable interface is an interesting technique to trick the virtual machine (VM). The principal mode of shareable interface attack [10] is to obtain type confusion without modifying the CAP files. To accomplish such an attack, the authors created two applets which are communicated using the shareable interface mechanism. Each applet uses a different type of array to exchange data in order to create the type confusion. In this case, the BCV cannot detect any problem during compilation or while loading the files. In our experience, most of the cards with an on-card BCV refused to allow applets using shareable interface mechanism. Since it is impossible for an on-card BCV to detect this kind of abnormality, Hubbers *et al.* [10] proposed the hypothesis of shareable interface.

Another type of attack is the abort transaction, which is very difficult to implement but it is a widely used concept in database system. The purpose of this transaction is to make a group of atomic operations. By definition, the roll-back mechanism should also deallocate any objects allocated during an aborted transaction and reset the references of such objects to null. However, the authors

found that some cases where the card keeps the reference to objects allocated during transaction, even after a rollback.

A fault model is explained in [16,17] to attack the smart card by physical means. Nowadays, it is also possible to obtain precise byte errors using a laser beam injection into the smart card memory or to perturb the runtime execution (like a pipeline modification). Due to this physical fault, a logical attack can occur which is called a combined attack. Barbu *et al.* proposed an attack based on the `checkcast` instruction [15] which is used to modify the runtime type by injecting a laser beam. Then this applet gets installed on the card after it has been checked by an on card BCV and it is considered to be a valid applet. The aim of this attack is to create a type confusion to forge a reference of an object and its contents. The authors also explained the principle of instance confusion, similar to the idea of type confusion where the objective is to confuse an instance of object A to an object B by inducing a fault dynamically by using a laser beam.

Bouffard *et al.* described in [13], two methods to change the control flow graph of a Java Card. The first one is EMAN 2, which provides a way to change the return address of the current function. This information is stored in the Java Card stack header. Once the malicious function exits during the correct execution, the program counter returns to the instruction which addresses it. The address of the JPC is also stored in the Java Card Stack header. An overflow attack success to change the return address by the address of the malicious byte code. Since there is no runtime check on the parameter, it allows a standard buffer overflow attack to modify the frame header.

4.4 Basic Events: Possible Countermeasures

For stack area protection, Girard mentioned in [18], that the system area of the frame is very sensible. So, he suggested to split the frame and manage the stack context separately from the stack and the locals of the frame. In [15], Barbu proposed a real approach by modifying the value of the security context in case of an illegal modification delegating the control to the firewall. He also suggested a stack invariant by XOR-ing the pushed and popped value. Dubreuil *et al.* [14] proposed to use a typed stack without paying the cost of two stacks manipulation. In the case of method area protection, it is possible to execute data as program, considering the value to be XOR-ed should be dependent to the type of the variables as suggested in [18]. Here, the data are protected within the same bounds of possibility. Barbu [15] proposed exactly the same approach, while Razafindralambo *et al.* [19] demonstrated that it is possible to recover the XOR value by using a dynamic syntax adding randomization to the XOR function. The implementation of all these countermeasures are during the linking step. In order to avoid the exploitation of the linker, an algorithm has been proposed in [14] to efficiently check the usage of the token to be linked by manipulating the reference location component.

Séré proposed three countermeasures in [20]. The first one is the bit field method based on the nature of the byte code that must remain non mutable (*i.e.* an instruction cannot be replaced by an operand). The second one is the

basic block method, which is an extension of the work described in [4], where a checksum is calculated during the execution of a linear code fragment and it is checked during the exit. The last one is the path check method, a completely different approach based on the difference between the expected execution paths and the current one. Any divergence or code creation can be detected by coding the path in an efficient way.

These countermeasures are of particular interest because they are addressing the two events of the first level of code integrity. The patent [21] describes a similar approach to the third countermeasure of Séré, but it is performed at the applicative level. A similar approach, based on flags and preprocessing is there to detect unauthorized modification in the control flow, as mentioned in [22]. To avoid the exploitation of the `checkcast` attack, an efficient implementation of this instruction was proposed in [15].

5 Definition of an API to Mitigate the Undesirable Events

Countermeasures can be implemented at two levels: virtual machine (system) or an applicative. The main advantage with system countermeasures is that the protections are stored in the ROM memory, which is a less critical resource than the EEPROM. So it is easier to deal with integration of the security data structures and code in the system. With applicative countermeasures, it is possible to implement several checks inside the application code to ensure that the program always executes a valid sequence of code on valid data. It includes double condition checks, redundant execution, counter, etc. and are well known by the developers. Unfortunately, this kind of countermeasures drastically increase the program size because, besides the functional code, it needs security code and the data structure for enforcing the security mechanism embedded in the application. Furthermore, Java is an interpreted language therefore its execution is slower than that with a native language. So, this category of countermeasures experiences bad execution time and adds complexity for the developer. But the main advantage here is that the developer has a precise knowledge of the sensitive part of his code. A new approach exists, which is driven by an application and it includes means in the system layer.

This solution presents a good balance between memory footprint and optimization of the countermeasure. This also presents a possibility to offer security interoperability between several manufacturers who implement this API.

5.1 Model of Attacks

Using the FTA analysis, we defined attack paths that allows a simple laser based fault injection. They correspond to the effect of a laser fault on the JVM. By refining the analysis, we discover three new possibilities:

- If the effect of laser affects the value returned by a method it can execute a code fragment without the adequate authorization or in a conditional evaluation it can change the branch which is to be executed. This attack targets

the Java Card evaluation stack where the value returned by the function is stored,

- The effect of fault can also bypass the execution of a given method. By modifying the JPC one can jump over a method that performs a security test letting on top of the stack the address of the called object. Of course, the stack has a high probability to be ill typed (non compatible) at the end of the current method. Nevertheless an attacker can still send high value information before being detected,
- It can also modify the address of data to be copied in the I/O buffer or change the number of byte to be copied. This attack allows to dump memory in an arbitrary way.

They have been introduced as a new intermediate node: JPC CORRUPTION and a new basic event: EVALUATION STACK CORRUPTION which can be obtained either by a logical attack or a fault attack. They can be found not only on different branches of the code integrity tree but also on the code confidentiality tree like the leaves of the intermediate node CONDITIONAL BRANCH, they are combined with an AND condition to generate the CODE DE-SYNCHRONIZATION intermediate node.

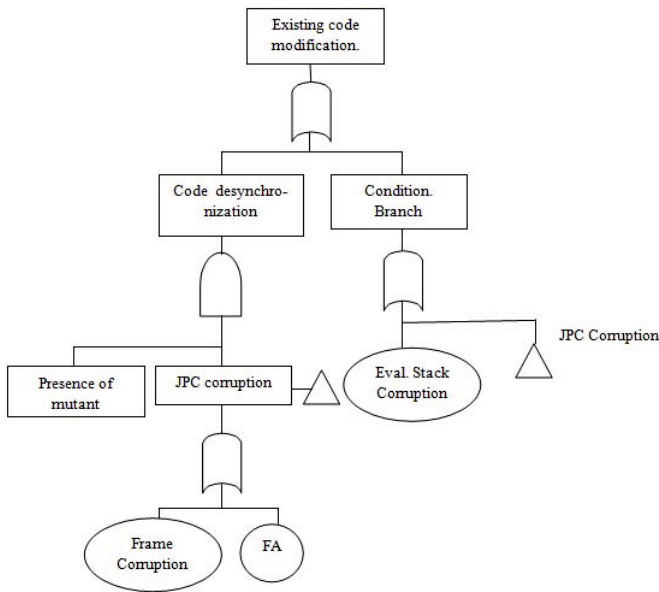


Fig. 5. JPC corruption and its effects

With the FTA analysis we were able to determine new potential vulnerabilities. The next step is to design efficient countermeasures to ensure a high coverage with a minimal memory footprint.

5.2 The INOSSEM API

Since several attacks can be performed against smart cards, an API has been defined, so that a given application can request to increase security for a code fragment. For example, to mitigate all the attacks against the stack, (*i.e.* the parameters of a called method), it is possible to invoke a method with a signature added as a parameter. Then, the application at the beginning of the execution of the called method should verify that the parameters have not been modified by a fault attack (listing 1.1).

```
// initialize the signature object with the first parameter
paramChkInstance.init(firstParam);
paramChkInstance.update(secondParam); //update signature buffer
short paramChk = paramChkInstance.doFinal(userPIN); // sign it
// invoke the sensitive method
sensitiveMeth(firstParam, secondParam, userPIN, paramChk);
```

Listing 1.1. Use of the `ParamCheck` class to sign parameters

All sensitive methods of the Java Card applet must ensure that no attack against the integrity of the frame occurred during the call. If the signature does not match, the card must be blocked, as shown in the listing 1.2

```
void sensitiveMeth (byte p1, short p2, OwnerPIN p3, short chk) {
// initialize the signature object with the parameters
paramChkInstance.init(p1); paramChkInstance.update(p2);
//... check the integrity of the locals in the frame
if (paramChkInstance.doFinal(p3) != chk)
    { // something bad occurred take some action }
```

Listing 1.2. Use of the `ParamCheck` class to verify parameters.

The second attack is related with JPC corruption. The control flow of the program can be modified due to the attack. For mitigating it, we choose to implement a security automaton into the API. The security automaton represents the security policy in terms of control flow checks. It is a partial representation of the control flow of the program, thus providing a redundancy that can be checked during the execution of the applet. For example, before decrementing the value of a counter the authentication step must have occurred earlier. So we included some states' modification into the code with a call to `setState()` method of the API. In the constructor of the applet, we have initialized a structure representing the security automaton. Each time a call to `setState()` occurs, the system checks if the current state allows the transition to the new state, if not it throws a security exception (listing 1.3).

```
private void debit(APDU apdu) {
this.setState(CRITICAL_SECTION); // transition to a new state
if (!pin.isValidated()) { //disallows if PIN is not verified
    this.endStateMachine(PIN_VERIFICATION_REQUIRED_STATE);
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED); }
```

Listing 1.3. The security automaton

There are also Java classes which are used to protect local variables against integrity and confidentiality violation. These form a coherent means to protect the application against the attack that the FTA analysis brought to the fore.

6 Conclusion and Future Works

A Fault Tree Analysis method was proposed and applied in the smart card domain which can be used for safety and vulnerability analysis. We have identified four major undesirable events and refined the analysis to reach till the basic events representing the effect of either laser attacks, logical attacks or their combination. During the analysis we brought forth three new attacks considering the effect of a single laser fault. And in the INOSSEM project the partners defined an API that provides protections against these attacks. With this strategy, a smart card developer can adapt the level of security of the JVM for a given code fragment of its applet. We are currently considering quantification of the probability for an attacker to reach his objective in a given time or the overall mean time for the attack to overcome it. This on going work uses the BDMP formalism [23] with the models designed in this paper.

Acknowledgements. This work is partly funded by the French project INOSSEM (PIA-FSN2 -Technologie de sécurité et résilience des réseaux).

References

1. Leveson, N.G.: Software Safety - What, Why And How? ACM Computing Surveys 16(2), 125–164 (1986)
2. Stamatis, D.H.: Failure Mode and Effect Analysis: FMEA from Theory to Execution. ASQ Press (1995)
3. Helmer, G., Wong, J., Slagell, M., Honavar, V., Miller, L., Wang, Y., Lutz, R.: Software fault tree and colored petri net based specification, design and implementation of agent-based intrusion detection systems. IEEE Transactions of Software Engineering (2002) (submitted)
4. Prevost, S., Sachdeva, K.: Application code integrity check during virtual machine runtime. US Patent App. 10/929,221 (2004)
5. Moore, A.P., Ellison, R.J., Linger, R.C.: Attack modeling for information security and survivability. Technical report, DTIC Document (2001)
6. Fronczak, E.: A top-down approach to high-consequence fault analysis for software systems. In: Proceedings of the Eighth International Symposium on Software Reliability Engineering, p. 259. IEEE (1997)
7. Byres, E.J., Franz, M., Miller, D.: The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems. In: International Infrastructure Survivability Workshop (IISW 2004). IEEE (2004)
8. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065 (2011)
9. GlobalPlatform: Card Specification. 2.2.1 edn. GlobalPlatform Inc. (2011)

10. Hubbers, E., Poll, E.: Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours. Technical Report NIII-R0438, Radboud University Nijmegen (2004)
11. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. *Journal in Computer Virology* 6(4), 343–351 (2009)
12. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) *CARDIS 2010*. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
13. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) *CARDIS 2011*. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
14. Dubreuil, J., Bouffard, G., Lanet, J.L., Iguchi-Cartigny, J.: Type classification against Fault Enabled Mutant in Java based Smart Card. In: *Sixth International Workshop on Secure Software Engineering (SecSE)*, pp. 551–556. Springer (2012)
15. Barbu, G.: On the security of Java CardTM platforms against hardware attacks. PhD thesis, Grant-funded with Oberthur Technologies and Télécom ParisTech. (2012)
16. Blömer, J., Otto, M., Seifert, J.P.: A new CRT-RSA algorithm secure against bellcore attacks. In: *ACM Conference on Computer and Communications Security*, pp. 311–320. ACM, Washington, DC (2003)
17. Wagner, D.: Cryptanalysis of a provably secure CRT-RSA algorithm. In: *ACM Conference on Computer and Communications Security*, pp. 92–97. ACM, Washington, DC (2004)
18. Girard, P.: Contribution à la sécurité des cartes à puce et de leur utilisation. Habilitation thesis, University of Limoges (2011)
19. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.-L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Alcaraz Calero, J.M., Thomas, T. (eds.) *SNDS 2012*. CCIS, vol. 335, pp. 185–194. Springer, Heidelberg (2012)
20. Al Khary Séré, A.: Tissage de contremesures pour machines virtuelles embarquées. PhD thesis, Université de Limoges, 123 Avenue Albert Thomas, 87100 Limoges Cedex (2010)
21. Akkar, M.L., Goubin, L., Ly, O., et al.: Automatic integration of counter-measures against fault injection attacks (2003), Pre-print found at <http://www.labri.fr/Person/ly/index.htm>
22. Abadi, M., Budi, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13(1), 4 (2009)
23. Piètre-Cambacédès, L., Bouissou, M.: Attack and Defense Modeling with BDMP. In: Kotenko, I., Skormin, V. (eds.) *MMM-ACNS 2010*. LNCS, vol. 6258, pp. 86–101. Springer, Heidelberg (2010)

Does Malware Detection Improve with Diverse AntiVirus Products? An Empirical Study

Ilir Gashi², Bertrand Sobesto¹, Vladimir Stankovic², and Michel Cukier¹

¹ A. James Clark School of Engineering, University of Maryland, College Park, MD, USA
{bsobesto, mcukier}@umd.edu

² Centre for Software Reliability, City University London, London, UK
{i.gashi, v.stankovic}@csr.city.ac.uk

Abstract. We present results of an empirical study to evaluate the detection capability of diverse AntiVirus products (AVs). We used malware samples collected in a geographically distributed honeypot deployment in several different countries and organizations. The malware was collected in August 2012: the results are relevant to recent and current threats observed in the Internet. We sent these malware to 42 AVs available from the VirusTotal service to evaluate the benefits in detection from using more than one AV. We then compare these findings with similar ones performed in the past to evaluate effectiveness of diversity with AVs. In general we found that the new findings are consistent with previous ones, despite some differences. This study provides additional evidence that detection capabilities are improved by diversity with AVs.

Keywords: Empirical study, Intrusion tolerance, Malware, Measurement techniques, Security, Security assessment tools.

1 Introduction

All systems need to be sufficiently reliable and secure in delivering the service that is required of them. Various ways in which this can be achieved in practice range from the use of various validation and verification techniques, to the use of software fault/intrusion tolerance techniques and continuous maintenance and patching once the product is released. Fault tolerance techniques range from simple “wrappers” of the software components [1] to the use of diverse software products in a fault-tolerant system [2]. Implementing fault tolerance with diversity was historically considered prohibitively expensive, due to the need for multiple bespoke software versions. However, the multitude of available off-the-shelf software for various applications has made the use of software diversity an affordable option for fault tolerance against either malicious or accidental faults.

Authors in [3] detailed an implementation of an AntiVirus (AV) platform that makes use of diverse AVs for malware detection. A similar architecture that uses diverse AV email scanners has been commercially available for several years [4]. Thus, architectural solutions for employing diverse AV detection engines are already known and even commercially deployed. Results from empirical evaluation of the effectiveness of diversity for malware detection are, however, much more scarce.

The following claim is made on the VirusTotal site [5]: “*Currently, there is no solution that offers 100% effectiveness in detecting viruses, malware and malicious URLs*”. Given these limitations of individual AVs, designers of security protection systems are interested in at least getting estimates of the possible gains in terms of added security that the use of diversity (e.g. diverse AVs) may bring for their systems.

Two of the authors of this paper have previously reported [6-8] results from a study on the detection capabilities of different AVs and potential improvements in detection that can be observed from using diverse AVs. In those studies we reported that some AVs achieved high detection rates, but none detected all the malware samples. We also found many cases of *regression* in the detection capability of the AVs: cases where an AV would regress from detecting the malware on a given date to not detecting the same malware at a later date(s). We reported significant improvements in the detection capability when using two or more diverse AVs. For example, even though no single AV detected all the malware in these studies, almost 25% of all the diverse 1-out-of-2 pairs of AVs successfully detected all the malware.

The results presented in [6-8] are intriguing. However, they concern a specific snapshot in the detection capabilities of AVs against malware threats prevalent in that time period (1599 malware samples collected from a distributed honeypot deployment over a period of 178 days from February to August 2008). In the security field the threat landscape changes rapidly and it is not clear to what extent these findings can be generalized to currently spreading malware. It is also not clear whether the diversity benefits reported in [6-8] are specific to that specific collection environment and time period, or whether they are consistent with other environments and time periods.

Our work is motivated by the following claim from Fred Schneider in [9]: “*Experimentation is the way to gain confidence in the accuracy of our approximations and models. And just as experimentation in the natural sciences is supported by laboratories, experimentation for a science of cybersecurity will require test beds where controlled experiments can be run.*” In this paper we present results of an empirical study about possible benefits of diversity with currently spreading malware and compare our findings with those reported in [6-8]. The main aim of our study is to verify the extent to which the findings previously reported are true with more recent malware. Consistent with the statement in [9], through experimentation and empirical analysis our goal is to gain confidence in the accuracy of the claims and help security decision makers and researchers to make more informed, empirically-supported decisions about the design, assessment and deployment of security solutions.

The results provide an interesting analysis of the detection capability of the respective signature-based components, though more work is needed for assessing full detection capabilities of the AVs. Also, the purpose of our study is not to rank the individual AVs, but to analyze the effectiveness of using diverse AVs.

The rest of the paper is organized as follows: Section 2 summarizes related work; Section 3 describes the data collection infrastructure; Section 4 presents the results of our study; Section 5 compares the results reported in this paper with the ones in [6-8]; Section 6 discusses the implications of our results on the decision making about security protection systems, and presents conclusions and possible further work.

2 Related Work

Studies which perform analysis of the detection capabilities and rank various AVs are very common. One such study, which provides analysis of “at risk time” for single AVs, is given in [10]. Several sites¹ report rankings and comparisons of AVs, though readers should be careful about the definitions of “system under test” when comparing the results from different reports.

Empirical analyses of the benefits of diversity with diverse AVs are much less common. Apart from our previous work [6-8] (we refer to the findings reported there when we compare them with the findings from this paper in Section 5) we know of only one other published study [3] that has looked at the problem. An initial implementation of the Cloud-AV architecture has been provided in [3], which utilizes multiple diverse AVs. The Cloud-AV uses the client-server paradigm. Each machine in a network runs a host service which monitors the host and forwards suspicious files to a centralized network service. This service uses a set of diverse AVs to examine the file, and based on the adopted security policy makes a decision regarding maliciousness of the file. This decision is then forwarded to the host. The implementation [3] handles executable files only. A study with a Cloud-AV deployment in a university network over a six month period is given in [3]. For the files observed in the study, the network overhead and the time needed for an AV to make a decision are relatively low. The authors acknowledge that the performance penalties could be much higher if file types other than just executables are examined, or if the number of new files observed on the hosts is high.

Apart from Cloud-AV, which is an academic prototype, commercial solutions that use diverse AV engines for file and e-mail scanning are also available².

3 Experimental Infrastructure

The malware have been collected on a distributed honeypot architecture using Dionaea - a low-interaction honeypot used to emulate common vulnerabilities, capture malicious payloads attempting to exploit the vulnerabilities and collect the binary files downloaded or uploaded. In summary, the main components of the experimental infrastructure and the process of data collection are as follows (full details are available in our technical report [11]):

- Dionaea has been deployed on 1136 public IP addresses distributed in six different locations in France, Germany, Morocco and the USA.
- The subnets do not contain the same number of IP addresses and the configuration differs between networks. Many IP addresses belong to the University of Maryland (where two of the authors of this paper are based). Note that neither all networks apply the same security policies nor are protected in the same way.

¹ av-comparatives.org, av-test.org/, virusbtn.com/index

² gfi.com/maildefense/,
pcworld.com/article/165600/G_data_internet_security.html

- We deployed the default configuration of Dionaea which exposes common Internet services such as http, ftp, smtp, MS SQL, MySQL, as well as Microsoft Windows and VoIP protocols. These services and protocols emulate known vulnerabilities and can trap malware exploiting them. Due to the types of vulnerabilities and protocols emulated, Dionaea mainly collects Windows Portable Executable (PE) files.
- For this study, one Dionaea instance was deployed on each separate subnet, running on a different Linux virtual machine. To facilitate the analysis, all the malware collected by Dionaea and the information relative to their submission were merged and centralized on a single server.
- Every day at midnight a Perl script downloads from the virtual machines running Dionaea the binary files and the SQLite database containing the malware capture information. This script then submits the whole malware repository to VirusTotal [5], which is a web service providing online malware analysis for various AVs. For each binary file successfully submitted, VirusTotal returns a scan key. The scan key is composed of the binary's SHA1 hash and the timestamp of the submission. To ensure a correct submission of each file and to later retrieve the analysis results the script keeps track of the scan keys.
- A second Perl script retrieves the reports for each malware using the scan keys generated by VirusTotal. For each malware, VirusTotal returns an array containing the number of AVs that have flagged the file as malicious, the total number of AVs used in the analysis, and a hash table mapping the AVs names and versions with the signatures name.

4 Results

4.1 Basic Statistics from Our Results

Using the dataset given in Section 3 we explore the potential benefits in malware detection from employing diverse AVs. We start with some descriptive statistics.

Data collection lasted for 23 days: 8-30 August 2012. During this period we collected 922 malware. These malware were sent to VirusTotal where they were examined by up to 42 AVs. We sent the malware on the first day of observation and continued to send them throughout the collection period. However the total number of data points is not simply $23 * 922 * 42$. It is smaller because:

- not all malware were observed in the first day of the collection – we continued to observe new malware throughout the collection period and we cannot send a newly collected malware to older versions of AVs running on VirusTotal;
- VirusTotal may not always return results for all AVs – we are not sure why this is. VirusTotal is a black-box service and its internal configuration is not provided. We presume that each AV is given a certain amount of time to respond; if it doesn't, VirusTotal will not return a result for that AV. Additionally a particular AV may not be available at the time we submit the malware for inspection.

A unique “demand” for the purpose of our analysis is a $\{\text{Malware}_j, \text{Date}_k\}$ pair which associates a given malware j to a given date k in which it was sent to VirusTotal. We treat each of the malware sent on a different date as a unique demand. If all 922 malware were sent to VirusTotal on each of the 23 days of data collection, then we would have $922 * 23 = 21,206$ demands. But as explained above, due to missing data, the number of demands sent to any of the AVs is smaller than 21,206.

If we now associate a given AV $_i$'s response to a given malware j on a given date k then we can consider each of our data points in the experiment to be a unique triplet $\{\text{AV}_i, \text{Malware}_j, \text{Date}_k\}$. For each triplet we have defined a binary score: 0 in case of successful detection, 1 in case of failure. Table 1 shows the aggregated counts of the 0s and 1s for the whole period of our data collection. We have considered as success the generation of an alert by an AV regardless of the nature of the alert itself.

Table 1. Counts of detections and failures for triplets $\{\text{AV}_i, \text{Malware}_j, \text{Date}_k\}$

<i>Value</i>	<i>Count</i>
<i>0 – detection / no failure</i>	<i>766,853</i>
<i>1 – no detection / failure</i>	<i>65,410</i>

4.2 Single AV Results

Table 2 contains the failure rates of all 42 AVs. The ordering is by the failure rate (second column) with the AV with the smallest failure rate appearing first.

The third column in Table 2 counts the number of “releases” of a given AV recorded by VirusTotal. We presume these are the versions of either the rule set or the release version of the detection engine itself. It seems that different products have different conventions for this. Amongst the 3 best AVs in our study, AntiVir reports only 1 release version whereas ESET-NOD32 and Ikarus have 36 and 27 respectively.

The fourth and fifth columns of Table 2 report about an interesting phenomenon first described in our previous work [8] – some AVs *regress* on their detection capability. That is, they detected the malware at first, and then failed to detect the malware at a later date(s), probably due to some updates in the respective AV's rule definitions. The fourth column contains the number of malware on which a given AV regressed, and the fifth column contains the number of instances of these regressions (since an AV may have regressed more than once on a given malware: alternated between detection and non-detection of a malware several times). We note that even a few AVs, which are in the top ten in terms of the overall detection rates, did have cases of regressions. Such a phenomenon can be due to various reasons. For instance, the vendor might have deleted the corresponding detection signature as a consequence of the identification of false positives associated to it, or they might be attempting to consolidate the signature based detection rules (i.e. define a smaller number of more generic rules) to achieve faster detection.

Table 2. Failure rates, number of releases and regression data for each AV

AV Name	Failure rate	Number of “releases” of the AV in VirusTotal	Count of Malware on which AV regressed	Count of Regression Instances
AntiVir	0.000049	1	0	0
ESET-NOD32	0.000049	36	0	0
Ikarus	0.000049	27	0	0
Kaspersky	0.000050	1	1	1
Sophos	0.000050	2	0	0
VIPRE	0.000098	29	0	0
McAfee	0.000099	1	0	0
Norman	0.000099	1	0	0
Emsisoft	0.000208	1	3	3
Symantec	0.001112	2	0	0
F-Secure	0.001204	1	0	0
Avast	0.001211	1	0	0
BitDefender	0.001232	1	0	0
PCTools	0.001244	1	0	0
Jiangmin	0.001286	1	0	0
AVG	0.001342	1	0	0
GData	0.001627	1	8	8
TrendMicroHouseC.	0.001847	2	13	13
K7AntiVirus	0.001881	19	0	0
McAfee-GW-Ed.	0.002232	1	43	43
VirusBuster	0.002256	27	0	0
nProtect	0.002503	26	0	0
Microsoft	0.002515	3	0	0
TheHacker	0.002522	1	1	1
TrendMicro	0.002530	2	26	26
DrWeb	0.003517	2	0	0
ViRobot	0.003518	1	0	0
Panda	0.003530	1	4	17
TotalDefense	0.003708	23	0	0
VBA32	0.006567	2	55	55
Comodo	0.009233	32	139	151
CAT-QuickHeal	0.010306	1	1	1
AhnLab-V3	0.010950	25	105	120
F-Prot	0.011789	1	1	1
CommTouch	0.012832	1	1	1
eSafe	0.165981	1	8	9
Rising	0.226335	18	10	10
SUPERAntiSpyware	0.356355	2	0	0
ClamAV	0.377830	1	0	0
Antiy-AVL	0.412142	1	1	1
Fortinet	0.670168	1	5	5
ByteHero	0.963825	1	33	59

Even though some of the AVs have very good detection rates, none of them have detected all the malware in our study. We can also see that some AVs have really low detection rates, with the bottom 7 AVs failing to detect more than 10% of all the demands sent to them. We are not certain why this is the case. It could be because the AV vendors are failing to keep their AVs in VirusTotal up to date with their latest signatures even if their products in commercial installations are up to date (though some of these vendors do seem to be updating their products with new release numbers, as evidenced from the values in the third column). Alternatively, it could be because these AVs genuinely have low detection rates for this dataset.

In our previous work [6] we have also observed a similar phenomenon: six of the AVs had failure rates higher than 10%. We cannot report if any of these 6 AVs are in the subset of the 7 AVs above, since in [6] the AV names were anonymised. Before we proceeded with the diversity analysis in [6] we discarded the AVs that had failure rates higher than 10%. We justified this by stating “*It is inconceivable that any system administrator would choose AVs that have such a bad detection rate to be used in their system. Hence we decided to remove the 6 worst performing AVs from further analysis. The decision was also influenced by our goal to make any results of the benefits of diversity appear fairer. A criticism that can be made if these 6 worst performing AVs are included in the analysis is that improvements in detection capability through the use of diversity will of course be higher if you have such poorly performing individual AVs in the mix. By removing them we make our estimates of the benefits of detection via diversity more conservative*”. To keep a consistent comparison of the results in this paper with those reported in [6-8] we discarded the AVs with failure rates greater than 10% from the diversity analysis. The rest of the analysis is based on the best 35 AVs from Table 2.

4.3 Diverse AV Results

To evaluate the possible benefits in detection capabilities that using diverse AVs may bring, we looked at two different types of diversity configurations/setup:

- *1-out-of-N* (or *1ooN*), where N is the total number of AVs in a given configuration – a malware is deemed to have been detected on a given date as long as at least one of the AVs detected it.
- *r-out-of-N* (or *rooN*) where N is the total number of AVs in a given configuration and *r* is the minimum number of AVs that must detect a malware on a given date for it to be deemed as detected. We looked at two particular voting options:
 - *majority voting*: where N is an odd number and *r* is $(N+1)/2$ - this allows a tie-breaker via majority voting;
 - *trigger level*: where a fixed number *r* of AVs have to detect a malware (i.e. “be triggered”) before the malware *is considered* to be detected – we will present results when the trigger level *r* equals 2 or 3.

Of course, many other voting configurations are possible, but we chose these two above as they best represent the contrasting tradeoffs between detection rates and false alarm rates that decision makers should take into consideration when deciding on a diverse system configuration: a *1ooN* configuration could be used to maximize the detection rates of genuine malware; a majority voting one could be used to curtail the false positive rate; the trigger level configurations allow a trade-off between these two. Note that since we are dealing with confirmed malware samples we cannot really present any data about false positives. However, the *rooN* results allow us to get initial estimates of how is the rate of correct detections of confirmed malware affected if one uses majority voting or trigger level setup to reduce the false positive rate.

Due to space limitations we only present a snapshot of the results we obtained. We concentrated on showing a graphical representation of the results. The details and the tables from which these graphs are generated as well as other detailed results are available in [11]. The cumulative distribution functions (*cdf*) of the failure rate achieved for *1ooN*, *2ooN*, *3ooN* and majority voting setups are shown in Figure 1.

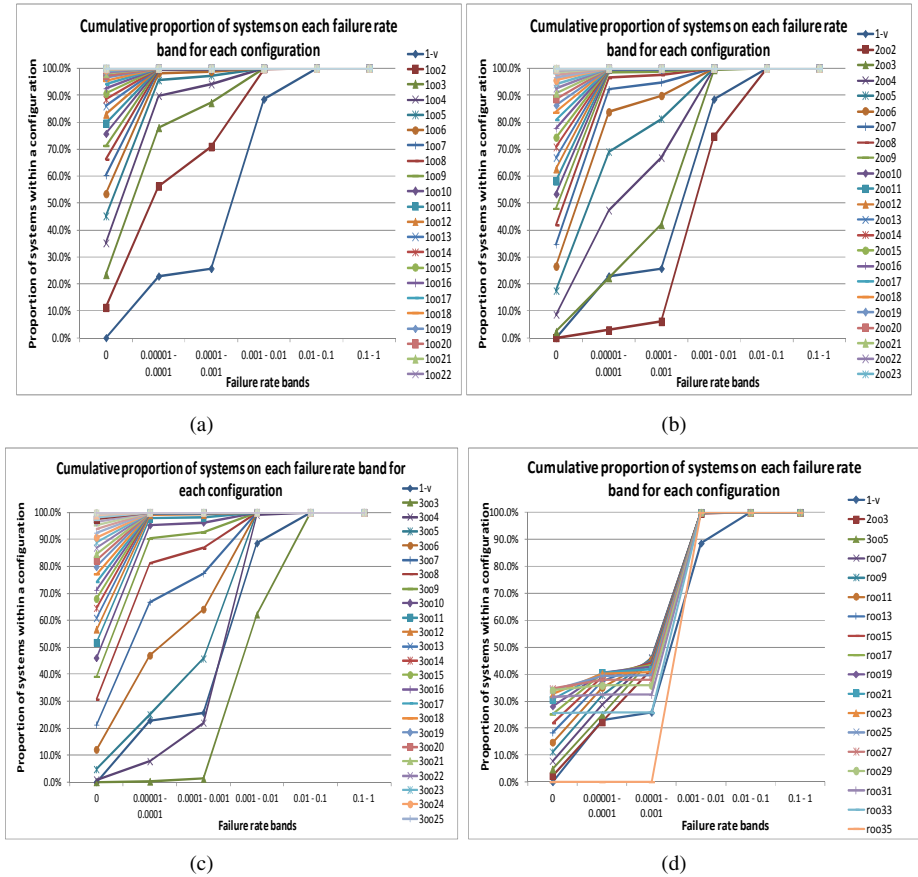


Fig. 1. Cumulative distribution of failure rate for a) $100N$, b) $200N$, c) $300N$, and d) $rooN$ setups

For part (a), (b) and (c) of Figure 1 the legend shows configurations up to $N = 22$, 23 or 25 , but in fact all $100N$, $200N$ and $300N$ diverse configurations are contained in the graph. For $100N$, $200N$ and $300N$ configurations the figures visualize the trend of improving failure rates as we add more AV products in diverse configurations. The shape of the distributions for the majority voting setup (part (d) of Figure 1) is different and we see a decline in the proportion of perfect systems after $N=29$. This is because majority voting setups, which operate with higher degrees of diversity, require more AVs (than $100N$, or trigger level setups) to detect a malware before raising an alarm. This requirement, in turn, can deteriorate the detection rate. The main insight is the performance of majority voting setups is not improved by adding further AVs.

Figure 2 presents the difference in the proportion of the non-perfect detection combinations in $100N$ and trigger level setups. We can see that to get 90% of all $100N$ systems to have a perfect detection of all malware in our dataset we need N to be roughly 15 (we can observe this in Figure 2 by following where the x-axis value of 15 crosses the y-axis value $1.E-01$ for the $100N$ line). To get 90% of $200N$ systems to detect all malware we need N to be roughly 20, whereas N for $300N$ is roughly 23.

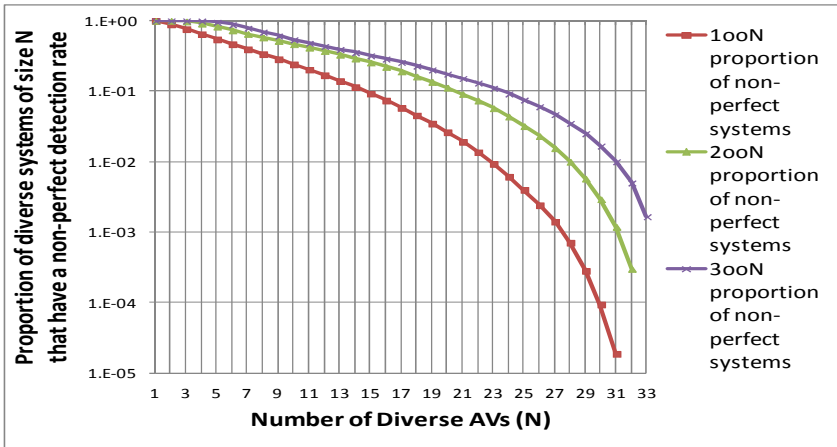


Fig. 2. Proportion of diverse systems of size N that have a non-perfect detection rate

This is another way in which an administrator could measure the added cost of seeking confirmation from 2 or 3 AVs before raising alarms.

5 Comparison of Results with the Earlier Study

We now discuss and compare the results of the study presented in this paper with those from our previous study [6-8]. In summary:

- In [6-8] despite the generally high detection rates of the AVs, none of them achieved 100% detection rate.
Our observation: we also observe this with the new dataset. Eight of the AVs in our study had failure rates smaller than $1.E-04$ but none of them detected all the instances of malware (on all days) in our study either.
- In [6-8] the detection failures were both due to an incomplete signature databases at the time in which the samples were first submitted for inspection, but also due to regressions in the ability to repeatedly detect malware as a consequence, possibly, of the deletion of some signatures.
Our observation: we also observe this with the new dataset. *Regressions* in the detection capabilities were observed even with AVs who are ranked in the top 10 in our dataset in terms of their detection capability.
- In [6-8] considerable improvements in detection rates were observed from employing diverse AVs: almost 25% of all the diverse pairs, and over 50% of all triplets in respective 1-out-of- N configurations successfully detected all the malware.
Our observation: we observe improvements in 1-out-of- N configurations though not as good as those in [6-8]: under 12% of all diverse pairs, and under 24% of all diverse triplets detected all the malware in 1-out-of- N configurations. To get over 50% of all combinations to detect all the malware we need 6 AVs rather than 3.

- In [6-8], no malware caused more than 14 AVs to fail on any given date. Hence perfect detection rate, with the dataset from [6-8], is achieved by using any 15 AVs in a 1-out-of-N configuration.
Our observation: We had malware that failed to be detected by up to 31 AVs in our dataset. So to get perfect detection rates with our dataset for all possible instances of a diverse configuration, we need 32 AVs.
- In [7] the detection rates were lower for “trigger level detection” (2-out-of-N and 3-out-of-N) and majority voting (r-out-of-N) setups compared with 1-out-of-N but on average are better than using a single AV.
Our observation: We confirm that these findings are consistent with what we found previously. Additionally we found that the proportion of perfect majority voting systems is considerably higher with the new dataset compared with those reported in [7]: in the new analysis more than 34% of possible combinations of 140027 and 150029 majority voting configurations achieved perfect detection. In [7] the proportion of perfect detection majority voting systems never reached more than 6% for any of the combinations in that study. This may have implications on the choice of diverse setups that administrators may use; especially if false positives become an issue. If the detection capability against genuine malware is not (significantly) hampered from a majority decision by diverse AVs, then majority voting setups could become more attractive to administrators than 100N configurations.
- In [6] significant potential gains were observed in reducing the “at risk time” of a system by employing diverse AVs: even in cases where AVs failed to detect a malware, there is diversity in the time it takes different vendors to define a signature to detect a malware.
Our observation: This is also supported by the results in our study (though due to space limitations we could not elaborate on this here; see [11] for details).
- In [6] an empirically derived exponential power law model proved to be a good fit to the proportion of systems in each simple detection (100N) and trigger level detection (200N and 300N) diverse setup that had a zero failure rate.
Our observation: we did not explore the modeling aspects in this paper. There do appear to be some differences between the shapes of the empirical distributions of the proportion of diverse systems of size N that have a non-perfect detection rate (Figure 2) from what is presented in [6]. The empirical distributions in Figure 2 do look like they follow an exponential power law model (possibly of a different form than in [6]), so further work is needed to check whether the model outlined in [6] applies to this dataset. A generic model would allow a cost-effective prediction of the probability of perfect detection for systems that use a large number of AVs based on measurements with systems composed of fewer (say 2 or 3) AVs.

6 Discussion and Conclusions

We reported analysis of empirical results about the possible benefits of diversity with currently spreading malware and compared and contrasted the findings with those we reported previously using a different dataset [6-8]. We verified the extent to which the findings from [6-8] are relevant with more recent malware. The new results were in

general consistent with those reported in [6-8], though there were differences. The consistent results were:

- None of the single AVs achieved perfect detection rates
- A number of the AVs *regressed* in their detection behavior (i.e. failed to detect malware which they had successfully detected in the past)
- Considerable improvements in detection capability when using diversity
- As expected, a decreasing ordering in the effectiveness of detection capability with 1-out-of-N, 2-out-of-N, 3-out-of-N and majority voting diverse setups is observed
- Using diverse AVs helps with reducing the “at risk time” of a system

The main differences in the results were:

- Despite significant improvements from diversity with 1-out-of-N, 2-out-of-N and 3-out-of-N, they are lower than those reported in [6-8]. For example with 100N we observed that 12% of all diverse pairs and 24% of all diverse triplets detected all the malware (compared with approximately 25% and 50% respectively in [6-8])
- On the other hand, we observe a much higher proportion of perfect detection majority voting systems in all setups compared with the results observed in [6-8]
- The shape of the empirical distribution of non-perfect detection systems as more diverse AVs are added seems different from that observed in [6]

The aim of this work is to provide more evidence on the possible benefits of using diverse AVs to help with malware detection and therefore help security decision makers, and the researchers in the field, to make more informed, empirically-supported decisions on the design, assessment and deployment of security protection systems.

The limitations to this work that require further research and prevent us from making more general conclusions are as follows. First, we have no data on false positives. Studying the detection capabilities with datasets that allow measurements of false positives in addition to false negative rates will allow us a better analysis of the tradeoffs between the various 1-out-of-N, trigger level and majority voting detection setups. Second, we have only tested the detection capability when subjecting the AVs to Windows portable executable files. Further studies are needed to check the detection capability for other file types e.g. document or media files. Third, the AV products contain more components than just the signature-based detection engine which is what the VirusTotal service facilitates. Further studies need to include the detection capabilities of these products in full. Fourth, due to lack of space in this paper, we have not explored the modeling aspects and modeling for prediction. Our next step is to do further work in checking whether a form of the power law distribution observed in [6] also applies with this dataset.

The current work in progress includes studying the patterns with which AVs label the malware when they detect them. We are interested in exploring the relationship between malware label changes and AVs *regressing* on their detection capability. In addition, we plan to investigate whether polymorphic malware is identified with the same labels by the AVs and whether this can aid with diagnosis and recovery from malware infections.

Acknowledgement. This work was supported in part by the City University London Pump Priming fund via grant “Empirical Analysis of Diversity for Security”. The authors thank Gerry Sneeringer and the Division of Information Technology at the University of Maryland for allowing and supporting their research.

References

1. van der Meulen, M.J.P., Riddle, S., Strigini, L., Jefferson, N.: Protective Wrapping of Off-the-Shelf Components. In: Franch, X., Port, D. (eds.) ICCBSS 2005. LNCS, vol. 3412, pp. 168–177. Springer, Heidelberg (2005)
2. Strigini, L.: Fault Tolerance against Design Faults. In: Diab, H., Zomaya, A. (eds.) Dependable Computing Systems: Paradigms, Performance Issues, and Applications, pp. 213–241. J. Wiley & Sons (2005)
3. Oberheide, J., Cooke, E., Jahanian, F.: Clouddav: N-Version Antivirus in the Network Cloud. In: The 17th USENIX Security Symposium, pp. 91–106 (2008)
4. GFI. Gfmaildefence Suite, <http://www.gfi.com/maildefense/> (last checked 2013)
5. VirusTotal. Virustotal - a Service for Analysing Suspicious Files, <http://www.virustotal.com/sobre.html> (last checked 2013)
6. Bishop, P., Bloomfield, R., Gashi, I., Stankovic, V.: Diversity for Security: A Study with Off-the-Shelf Antivirus Engines. In: The 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE 2011), pp. 11–19 (2011)
7. Bishop, P.G., Bloomfield, R.E., Gashi, I., Stankovic, V.: Diverse Protection Systems for Improving Security: A Study with Antivirus Engines. City University London, London (2012)
8. Gashi, I., Leita, C., Thonnard, O., Stankovic, V.: An Experimental Study of Diversity with Off-the-Shelf Antivirus Engines. In: The 8th IEEE Int. Symp. on Network Computing and Applications (NCA 2009), pp. 4–11 (2009)
9. Schneider, F.: Blueprint for a Science of Cybersecurity. *The Next Wave* 19(2), 47–57 (2012)
10. Sukwong, O., Kim, H.S., Hoe, J.C.: Commercial Antivirus Software Effectiveness: An Empirical Study. *IEEE Computer* 44(3), 63–70 (2011)
11. Cukier, M., Gashi, I., Sobesto, B., Stankovic, V.: Technical report: Does Malware Detection Improve with Diverse Antivirus Products? An Empirical Study (2013), <http://www.csr.city.ac.uk/people/ilir.gashi/SAFECOMP2013/>

Software Fault-Freeness and Reliability Predictions

Lorenzo Strigini and Andrey Povyakalo

Centre for Software Reliability,
City University London, UK
{L.Strigini, A.A.Povyakalo}@city.ac.uk
<http://www.csr.city.ac.uk>

Abstract. Many software development practices aim at ensuring that software is correct, or fault-free. In safety critical applications, requirements are in terms of probabilities of certain behaviours, e.g. as associated to the Safety Integrity Levels of IEC 61508. The two forms of reasoning – about evidence of correctness and about probabilities of certain failures – are rarely brought together explicitly. The desirability of using claims of correctness has been argued by many authors, but not been taken up in practice. We address how to combine evidence concerning probability of failure together with evidence pertaining to likelihood of fault-freeness, in a Bayesian framework. We present novel results to make this approach practical, by guaranteeing reliability predictions that are conservative (err on the side of pessimism), despite the difficulty of stating prior probability distributions for reliability parameters. This approach seems suitable for practical application to assessment of certain classes of safety critical systems.

Keywords: Correctness, survival probability, conservative bounds, software safety standards.

1 Introduction

For critical applications of computers, it is important to demonstrate low enough likelihood that design faults (in particular software faults) cause, in operation, failures with severe consequences. A form of evidence that can support such claims is experience from either previous operation, in the same conditions as those for which the demonstration is needed, or “operational” testing that reproduced those conditions. Such evidence is often not offered in a suitable form.

Other forms of evidence are usually provided, often generically called “process” (or indirect) evidence: that methods believed to reduce the risk of defects were applied in development and verification and validation; and that the faults of concern are indeed likely to be absent (e.g., certain properties have been formally proved, stringent inspection or testing methods failed to detect faults, any fault revealed and considered important was fixed). Such process precautions are required by most standards for safety-critical (and security-critical) software.

Indeed, the process evidence required by a standard for a given criticality of the software’s functions is often the *only* evidence brought to support a claim

that the software will fail with acceptably low probability. But this use of the evidence is not supported by current software engineering knowledge [1, 2]. The methods documented are considered by their proponents to reduce the *likelihood of faults*; but we have really no scientific bases for claiming that a specific set of precautions will achieve a *failure rate*, or a probability of failure per demand, below a specific threshold as required by the system of which the software will be a part (e.g. a SIL level in IEC61508 [3]). Even if these methods do reduce the number of faults, fault numbers or fault densities are not sufficient to enable estimation of a failure rate or *pdf*.

There is a use of “process” evidence that *could* directly support claims of low probability of failure: as supporting belief in *absence of faults*. The goal of the process precautions is to avoid faults; and for products that are inherently simple, this goal might be achieved. Software about which a verifier concludes, by detailed analysis and/or proof, that it is correct, sometimes *is* correct. A fault-free software product has zero failure probability, over any duration of operation. We cannot generally claim to know that a software product is fault-free with certainty; but we could bring convincing evidence about a probability of it being so, and this probability may be high enough to help in proving that the risk in operation is low enough. A formal introduction to this approach and more complete arguments in its favour were given many years ago [4]; we return to it to propose a concrete approach to its application. Advantages of reasoning this way would include:

- the probability of $pdf = 0$ is a lower bound on the software’s probability of failure-free behaviour over any arbitrarily long period of operation (a serious advantage when making predictions for long-lived systems).
- while probabilities of failure per demand, or failure rates, depend on the frequencies of the various demands in the environment of use (the operational profile) of the system, a claim about probability of absence of faults would accompany a product to each new use for which the range of demands and the required responses are the same. Such relatively environment-independent claims would for instance be useful for the practice of “Safety Element out of Context” described in ISO26262 [5].

Many different words are used for properties similar to what we discuss, saying e.g. that the software is “correct”, or “free from faults” (or “from defects”), or “perfect”. We choose the term “fault-freeness”. Independently of the name used, to avoid logical fallacies one needs to apply this term carefully according to the context. We are interested in safety critical, software-based items; then, we will mean by “software faults” those that would cause behaviours that violate safety requirements when the software is used in the context of interest; and by “failures” those with respect to these safety requirements. Similar restrictive definitions could be applied for the case of software that is critical for security. Different definitions will apply in different contexts. For instance, a subcontractor may wish instead to demonstrate that the software it delivers satisfies the written specifications in its contract, irrespective of whether they are correct, and thus define “faults” and “failures” with respect to these specifications.

Assessments of the probability of operating for a period of time without safety failures are formally reliability predictions (reliability with respect to that subset of possible failures). In our mathematical treatment, we assume a system for which the reliability parameter is a probability of failure per demand (*pdf*); our approach can easily be extended to systems for which the reliability model is an exponential, continuous-time model, with a failure rate as its parameter.

In the rest of this paper, Section 2 examines how to use a claimed probability of fault-freeness P_p towards claims of actual interest, namely probability of failure-free operation (reliability) over prolonged operation (possibly a system's lifetime). Section 3 discusses how to integrate via Bayesian inference evidence from failure-free operation to improve the claimed reliability; it shows how to avoid the crucial difficulty of choosing a prior distribution and obtain predictions that are guaranteed to be conservative (not to err on the side of optimism). Section 4 positions our contribution with respect to other past and ongoing work on related approaches. Last, Section 5 examines how a claim of a certain P_p can be supported, and addresses the crucial issue that absolute certainty of fault-freeness can never be achieved, even for a product that is indeed fault-free. The last section discusses the value of the reported results and future work.

2 Reliability Predictions Using a Probability of Fault-Freeness

An advantage of reasoning with claims of fault-freeness is that they define lower bounds on long-term reliability, irrespective of the use (demand profile) to which the item will be subjected.

Reliability predictions based on a claimed probability of fault-freeness take a simple form: given a probability P_p of fault-freeness, the reliability of the item at any future time t , $R(t)$, satisfies $R(t) \geq P_p$. Thus, in particular, being able to claim a reasonably high P_p is a desirable option for systems with an operational life of many demands but that will not receive operational testing over a comparable number of demands. For instance, let us imagine a system with an intended lifetime of 10,000 (statistically independent) demands. To be 90% sure that it will not suffer any accident due to the software, we would need to demonstrate $pdf \leq 10^{-5}$; but we would get at least the same confidence if we could claim a 90% probability of fault-freeness.

We expect that claims based solely on probability of fault-freeness would not be accepted in many application domains: users, regulators and the general public would want to know some bound on the probability of failure for the case that the software *has* faults. But such confidence bounds on the *pdf* can be obtained from past operation or operational testing.

For instance, if we had 90% confidence that an item of avionic software has no faults such as to cause catastrophic failure, this by itself satisfies the regulatory requirement that catastrophic failures due to this equipment be “not anticipated to occur during the entire operational life of all airplanes of one type”, “usually expressed” as “probability on the order of 10^{-9} or less” per flight-hour [6]

(the often-quoted “ 10^{-9} ” requirement which has been forcefully argued to be infeasible to demonstrate statistically [1, 7]). Yet, the possibility that, in the unlikely (10% probability) case of faults being present, such faults cause a high probability (say 1%) of failure per flight, would probably seem unacceptable: some evidence would be required that even if faults are present the *pdf* would still be low. But this evidence would not need to demonstrate the 10^{-9} requirement. A much more modest claim, together with the probability of fault-freeness, would ensure a forecast of low enough risk during the early life of the aircraft type; and as operation continues, failure-free operation would increase the likelihood that the software is indeed fault-free, or if not, that its probability of failure is indeed very small.

We proceed to discuss this combination of evidence from operation or operational testing with probability of fault-freeness.

3 Inference from Operation or Testing

We examine now how to improve a reliability claim that includes probability of fault-freeness by adding evidence from operation or testing, if no failures (of the failure types of interest) have been observed.

Bayesian inference from operational testing is well understood. The unknown *pdf* is seen as a random variable, which we will call Q , with a *prior* probability density $f_Q(q)$. After observing success on t_{past} independent demands, the *posterior* probability of surviving t_{fut} further demands is:

$$R(t_{fut}|t_{past}) = \frac{\int_0^1 (1 - q)^{t_{past} + t_{fut}} f_Q(q) dq}{\int_0^1 (1 - q)^{t_{past}} f_Q(q) dq} \tag{1}$$

According to the previous discussion of probability of fault-freeness, the prior distribution for the unknown *pdf* has the form

$$R(t_{fut}|t_{past}) = f_Q(q) = P_p \delta(q) + (1 - P_p) f_{Q_n}(q) \tag{2}$$

where $\delta(q)$ is Dirac’s delta function; $f_{Q_n}(q)$ is itself a probability density function, for the random variable “value of the system *pdf* conditional on $pdf > 0$ ”.

After observing t_{past} failure-free demands, the posterior reliability is

$$\begin{aligned} & \frac{\int_0^1 \left((1 - q)^{t_{past} + t_{fut}} (P_p \delta(q) + (1 - P_p) f_{Q_n}(q)) \right) dq}{\int_0^1 \left((1 - q)^{t_{past}} (P_p \delta(q) + (1 - P_p) f_{Q_n}(q)) \right) dq} \\ &= \frac{P_p + (1 - P_p) \int_0^1 (1 - q)^{t_{past} + t_{fut}} f_{Q_n}(q) dq}{P_p + (1 - P_p) \int_0^1 (1 - q)^{t_{past}} f_{Q_n}(q) dq} \end{aligned} \tag{3}$$

We can describe the operation of Bayesian inference as reducing the values of the probability density function more for those values of *pdf* that are less likely to be true in view of the observed failure-free operation. Thus seeing no failures reduces the values of the probability density function for high values of *pdf*, and shifts probability mass towards the origin (towards $pdf = 0$).

3.1 Worst Case Prior Distributions and Worst-Case Reliability

A common problem in applying Bayesian inference is choosing a prior distribution for the unknown pdf , Q . Arguing from the pdf values observed for similar software will be difficult: there has been no systematic data collection activity that would allow this. “Expert judgement” tends then to be used. But all scientific evidence is that experts’ judgement of probabilities tends only to be good for phenomena of which they have actual experience of prediction followed by feedback about its accuracy: textbook examples of observed good probability prediction ability are weather forecasters and horse-racing bookmakers, who on a daily basis assign probabilities to events and very soon observe whether the event occurs or not. On this basis, for an expert in critical software to be accurate in assigning a probability density function for a product’s pdf would be unusual. We can expect an expert’s direct experience to include comparatively few products, hardly any examples of pdf close to 1, and for those with high reliability, insufficient information to judge their true value of pdf .

But for safety it is usually acceptable to demonstrate *pessimistic* predictions. We can then look for a *worst case* prior distribution that one could assume for the inference. We can show that such a worst case does exist, as follows. Consider a probability density function for the unknown pdf , Q , made of two probability masses: a mass P_p in $Q = 0$ and a mass $(1 - P_p)$ in $Q = q_N$. Now, if we assume q_N to be close to either end of the interval $[0, 1]$, reliability predictions after observing failure-free demands will be very high. Indeed, in the two limiting cases, predicted reliability will be 1: if $q_N = 1$, one test is enough to show that $P(Q = q_N) = 0$ and thus $Q = 0$ with certainty; and if $q_N = 0$, $Q = 0$ with certainty to start with. In between these extreme values, successful tests will reduce $P(Q = q_N)$ and increase $P(Q = 0)$, but still leave a non-zero probability of future failure. Thus, posterior reliability as a function of q_N is highest at the two ends of the interval, and must have a minimum somewhere in between.¹

A proof of existence of this worst-case prior distribution of pdf has two steps:

1. as a consequence of the Lemma proved in Appendix A, of all the prior distributions with a probability mass in $Q = 0$, the worst-case one is indeed a two-point distribution as above

$$P_s \delta(q) + (1 - P_s) \delta(q - q_N) \quad (4)$$

¹ These considerations highlight another important point: a prior that is *pessimistic* in terms of the reliability it implies may produce *optimistic* inference. Here, moving q_N closer to 1 implies, before failure-free operation, pessimism: a system likely to fail in few demands from the start of operation. But then observing it *not* failing over even few demands then logically makes it very likely to have 0 pdf (optimism). Which prior distributions will produce pessimistic posteriors depends both on which posterior prediction we wish to minimise (e.g. posterior reliability for t_{fut} demands vs posterior probability of fault-freeness) and on the specific observations (here, the number of failure-free demands). It would thus be wrong to take from the worst-case posterior distribution we obtain here any measure different from posterior reliability for t_{fut} demands, e.g. a certain percentile, or a posterior probability of fault-freeness, and believe it to be a conservative value for use in further claims about this system.

so that the posterior reliability (3) has the form

$$\frac{P_p + (1 - P_p)(1 - q_N)^{t_{past} + t_{fut}}}{P_p + (1 - P_p)(1 - q_N)^{t_{past}}} \quad (5)$$

2. among all such two points distribution, we can identify a value of $q_N \in (0, 1)$ that yields the lowest posterior reliability. Thus Bayesian inference can be applied on the basis of only P_p and N , removing the major obstacle of assessing the whole $f_{Q_N}(q)$ distribution.

Fig. 1a shows worst case posterior reliability as a function of the ratio t_{fut}/t_{past} , for various values of the prior probability of fault-freeness, P_p .

We can read this figure in various ways:

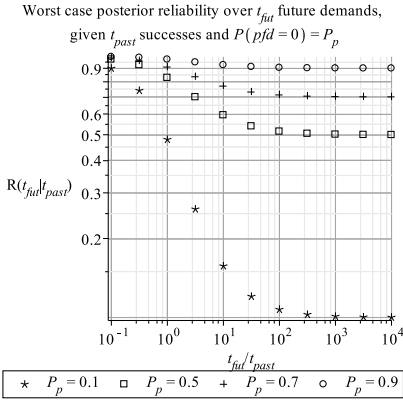
- given a certain amount of observed failure-free operation, the worst-case reliability predicted for a comparable amount in the future is satisfactorily higher than P_p . So, for instance, given a prior probability of fault-freeness $P_p = 0.5$ for a safety critical system in — say — a car model, after observing failure-free operation of a car fleet using that system for one year, the worst-case probability of failure-free operation for another year, for constant fleet size, is above 80%. Given $P_p = 0.9$, it would be more than 95%. However, as the prediction extends further and further into the future, the statistical evidence becomes less and less adequate for confident prediction, and the worst-case reliability asymptotically falls back to P_p as t_{fut} tends to infinity;
- given the horizon t_{fut} over which we want to predict reliability, the plot shows the number of t_{past} observations that we need to reach for the worst-case prediction to hit our intended target. For instance, expecting a safety protection system to have to face 100 demands over its lifetime, $P_p = 0.9$ and statistical testing over 1000 demands will give probability of going through this lifetime without failures upwards of 95%. More detail for scenarios with very high required worst-case reliability, and extensive operational testing, is given in Fig. 1b; the y -axis represents the probability of at least one failure over t_{fut} demands. If the number of test demands is much greater than the number of demands over the intended operational lifetime², even modest values of P_p give substantial confidence in failure-free operational life.

A special case of reliability prediction is the reliability for $t_{fut} = 1$, i.e. the system *pdf*. Fig. 1d shows the number t_{past} of failure-free demands one needs to observe to achieve a desired value for the worst case posterior *pdf*.

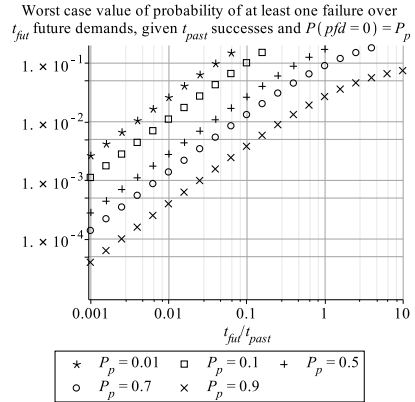
4 Related Work

In computer science there has been for a long time an opinion sector opposing the very idea of software reliability assessment, on the grounds that software can be made, and thus ought to be made, correct: 100% reliable.

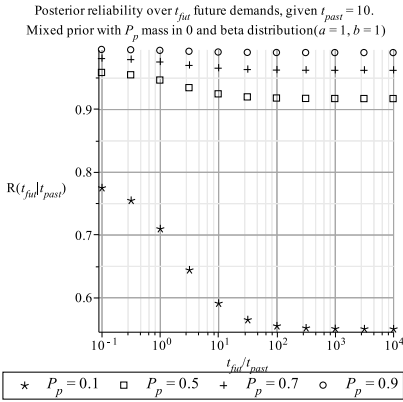
² An example of current interest concerns plans for testing of the protection system of the European Pressurised Reactor <http://www.hse.gov.uk/newreactors/reports/step-four/final-res-plans/resolution-plan-gi-ukepr-ci-02.pdf>



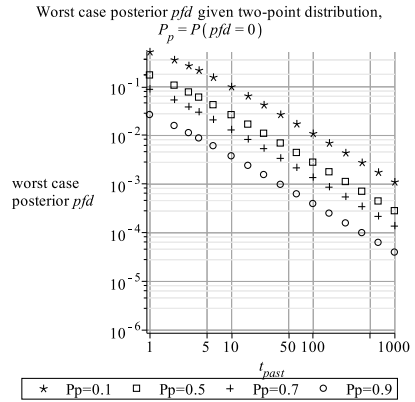
(a)



(b)



(c)



(d)

Fig. 1.

Several authors [8, 4, 9] have argued for the usefulness of estimating a probability of correctness. Voas and co-authors [8] argued that given a lower bound on the pfd that any one fault can produce (“testability”), one can infer from failure-free testing a probability of correctness. Bertolino and Strigini published the inference procedure for this case [10, 11]; however, one cannot usually demonstrate a lower bound on the probability of failure; it is hard to demonstrate experimentally that an event that is very unlikely to start with, and never observed, is actually impossible. Thus the treatment used here, producing a worst-case prediction without such assumptions, is a major step towards practical applicability.

The approach of accepting an incomplete description of prior pfd distributions, and obtaining conservative predictions by finding – among those prior distributions that match this description – the one that, combined with the actual

observations, would yield the worst posterior value for some reliability measure has been studied: (1) for reliability given a mean *pdf* [12]; (2) for the mean posterior *pdf*, given, about the prior distribution of the *pdf*, (2a) a confidence level on an upper bound [13], or (2b) some combinations among a probability of *pdf*= 0, a confidence level on an upper bound and a certain upper bound [14].

Littlewood and Rushby showed how claims of “perfection” can aid assessment of fault-tolerant systems made of diverse redundant components [15, 16].

5 Evidence for Fault-Freeness and “Quasi-Fault-Freeness”

Confidence in fault-freeness depends on “process” evidence being backed by arguments about why that process produces fault-free products with high enough probability. One might for instance reason that:

1. the current product is obtained by a process by which the same organisation produced a series of previous products (similar in their general requirements and complexity), that have proved to be fault-free; or
2. the verification steps that show this product to be correct (e.g., proof) are known by experience to catch a certain, high percentage of all faults, and together with estimates of the fault numbers to be expected before verification, this yields a probability of having *no* faults [17, 18, 19].

There are important difficulties. The relevance of past experience – whether the current product is somehow an anomaly in the “population” of the past products considered – is never certain. This is just the underlying difficulty of all statistically-based prediction, and we need not discuss it for this specific case. But here, the past experience itself is ambiguous: in argument 1 above, we cannot know *with certainty* that such past products were fault-free, but at most that they were scrutinised in many ways, operated for a long time without failures or problems that would cast any doubts on their correctness, and so on. As for argument 2, it assumes that in past experience we reached certainty about the number of faults in a product, and again such absolute certainty is impossible.

We outline here how this difficulty can be overcome in principle. Regarding for instance case 1 above, we consider that, if a past product successfully underwent stringent scrutiny and a long operational life, we cannot declare it fault-free with certainty, *but* it has a posterior distribution of *pdf* where most of the probability mass is either in 0 or close to it. The Lemma in appendix A shows that for worst-case reliability prediction, such a distribution can be substituted with a single probability mass in a point q_S close to 0. Similar experience for multiple similar products will also give confidence, say a probability P_s , that the same property applies to the current product. So, for the current product, we can use a pessimistic probability density function of *pdf* similar to equation 4:

$$P_s \delta(q - q_S) + (1 - P_s) \delta(q - q_N) \quad (6)$$

(where q_N accounts for the possibility that the *pdf* of the current system is *not* as low as that of the previous systems, and could be much worse); and find

a worst-case value of q_N , as in section 3.1. For q_S close enough to 0 (in view of t_{past} and t_{fut}), again the confidence P_s , together with worst case reasoning, could give useful reliability predictions, guaranteed to be conservative.

We have outlined this solution as a chain of reasoning steps: obtaining confidence in low *pdfs* of past products; then confidence that similar confidence in a low *pdf* applies to the new product; then using the latter for worst-case inference. This entire chain could instead be formalised in a single Bayesian model, in which all similar products have *pdf* distributions with parameters belonging in their turn to a common distribution, about which the operation of each product gives some information. Such models have been studied, e.g. [20]. A natural next step in this research is to apply them to the current problem, and check the feasibility of their use in concrete assessment and certification contexts.

6 Discussion and Conclusions

We have shown a way of using the evidence usually collected for assurance about safety critical systems, together with experience from operation or realistic testing, to achieve reliability predictions that can be proved to be conservative.

This relies on (i) using the process evidence to support a claim of “probability of fault-freeness”; (ii) applying Bayesian inference from the observation of failure-free operation and (iii) given strong uncertainty about the prior distributions to use, applying worst-case reasoning.

This approach reduces the impact of important difficulties with the current ways of stating quantitative claims about software failure in critical applications: the lack of scientific bases for deriving claims about *pdf* from the kinds of favourable evidence usually produced about such software; the difficulty of achieving enough operational or test experience to demonstrate very high reliability over long lifetimes; last, the difficulty of choosing convincing prior distributions for Bayesian inference is obviated by the ability to do worst case inference.

The predictions thus obtained will not always be as high as desired, for at least two possible reasons: (i) the evidence may simply not be strong enough (not enough operational experience, not strong enough prior probability of fault-freeness) to warrant as high a predicted long-term reliability as we seek; (ii) these methods are intentionally biased towards conservatism; they avoid the risk of erroneous optimism by accepting potential errors in the direction of pessimism. Of course, choosing to err in the direction of pessimism is a two-edged sword; it avoids dangerous errors but may make the prediction useless. By way of comparison, we show in Fig. 1c the posterior reliability obtained by assuming, together with a certain probability P_p of fault-freeness, that the *pdf*, if not zero, has a uniform distribution. This distribution might indeed be chosen as an “ignorance prior”, when one does not know what prior to believe, and seem reassuringly conservative because *before observing failure-free demands*, it is indeed very pessimistic: it means that if there are faults the expected *pdf* is 0.5. But this prior conservatism is deceptive. It implies that observing failure-free demands very quickly builds up confidence in future reliability: comparing Fig. 1c with Fig. 1a

(and noting the different vertical scales in the two plots) shows how far this assumption is from being conservative. Thus, if one has strong reasons to support a specific prior distribution conditional on non-fault-freeness, ($f_{Q_n}(q)$), by all means one should use it; but our worst case reasoning will illustrate how much of the predicted reliability hinges on believing that specific prior.

When performing inference using prior probability of fault-freeness (or of “quasi-fault-freeness”), failure-free operation gradually builds up confidence (increases predicted reliability). But observing even a single failure will radically undermine this confidence: the posterior probability of $pdf=0$ becomes zero; not being sure about our prior distribution for pdf when not zero ($f_{Q_n}(q)$), and wishing to be pessimistic, we must conclude that the pdf is indeed very high. Some may object to this apparent “fragility” of the approach. We contend that it is an advantage: it represents correctly the way confidence is gained for many critical products. Indeed, if a product that was reputed to be practically immune from design faults suffers a possibly design-caused failure, a normal reaction is to take it out of service, find the design fault and fix it (creating a new product and a new prior distribution of pdf); or demonstrate that the failure was not due to a design fault; or that the design fault exists but brings an acceptably low pdf in operation. In any case, the previous argument is discarded when the failure undermines the belief in a pdf so low that the probability of seeing any failure is also low. Our Bayesian formalisation faithfully represents this effect.

We strongly believe that this approach can improve the way that critical software-based systems are assessed. However, we acknowledge that we advocate the use of general evidence about the effectiveness of development methods that is not widely available. For the time being, this approach may be useful to organisations with strong internal data collection processes: they may well have enough evidence to build arguments that they will consider sound for their own risk assessments, or might support a claim made to a client or a safety regulator. A safety regulator may use our kind of worst-case reasoning to compare against the predictions that it has to judge in order to approve or reject a claim that a system is safe enough for operation. A company for which wrongly optimistic reliability predictions bring large economic risks (e.g. an automobile manufacturer, for which a doubt of a possible safety-critical fault may lead to massive recalls) may use this approach to assess its own risk, both prior to deployment and at any point in the operational life of its products.

A straightforward extension of this approach is to the case of continuous reliability with the unknown parameter being a failure rate λ instead of a pdf .

Another important question for further investigation is whether more complex Bayesian models, taking into account — for instance — experience in comparable products as in [20] can prove useful in practice, as suggested in Section 5, to ensure sound inference from “quasi-fault-freeness” of past products.

Acknowledgements. This research was supported in part by the Leverhulme Trust via project UnCoDe (“Uncertainty and confidence in safety arguments: effect on expert decision makers”) and by the SeSaMo project (“Security and Safety Modelling”), supported by the Artemis JU, and the United Kingdom

Technology Strategy Board (ID 600052). Our colleagues Bev Littlewood and David Wright contributed useful criticism to previous versions of this work.

References

- [1] Littlewood, B., Strigini, L.: Validation of ultra-high dependability for software-based systems. *CACM* 36(11), 69–80 (1993)
- [2] Littlewood, B., Strigini, L.: ‘Validation of ultra-high dependability...’ - 20 years on. *Safety Systems, Newsletter of the Safety-Critical Systems Club* (May 2011)
- [3] (IEC) International Electrotechnical Commission, IEC 61508, functional safety of electrical/ electronic/programmable electronic safety related systems
- [4] Bertolino, A., Strigini, L.: Assessing the risk due to software faults: estimates of failure rate vs evidence of perfection. *Software Testing, Verification and Reliability* 8(3), 155–166 (1998)
- [5] ISO, ISO 26262 road vehicles – functional safety (2011)
- [6] FAA, Federal aviation regulations far 25.1309. Advisory Circular AC 25.1309-1A, Federal Aviation Administration (1985)
- [7] Butler, R., Finelli, G.: The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE TSE* 19(1), 3–12 (1993)
- [8] Voas, J., Michael, C., et al.: Confidently assessing a zero probability of software failure. *High Integrity Systems* 1(3), 269–275 (1995)
- [9] Howden, W., Huang, Y.: Software trustability analysis. *ACM TOSEM* 4(1), 36–64 (1995)
- [10] Bertolino, A., Strigini, L.: On the use of testability measures for dependability assessment. *IEEE TSE* 22(2), 97–108 (1996)
- [11] Bertolino, A., Strigini, L.: Acceptance criteria for critical software based on testability estimates and test results. In: *SAFECOMP 1996*, pp. 83–94. Springer (1996)
- [12] Strigini, L.: Bounds on survival probabilities given an expected probability of failure per demand. *DISPO2 Project Technical Report LS-DISPO2-03*, Centre for Software Reliability, City University London (July 2003)
- [13] Littlewood, B., Povyakalo, A.: Conservative bounds for the pfd of a 1-out-of-2 software-based system based on an assessor’s subjective probability of ‘not worse than independence. *CSR Technical Report*, City University London (2012)
- [14] Bishop, P., Bloomfield, R., et al.: Toward a formalism for conservative claims about the dependability of software-based systems. *IEEE TSE* 37(5), 708–717 (2011)
- [15] Littlewood, B.: The use of proof in diversity arguments. *IEEE TSE* 26(10), 1022–1023 (2000)
- [16] Littlewood, B., Rushby, J.: Reasoning about the reliability of diverse two-channel systems in which one channel is ‘possibly perfect’. *IEEE TSE* 38(5), 1178–1194 (2012)
- [17] Shimeall, T., Leveson, N.: An empirical comparison of software fault tolerance and fault elimination. *IEEE TSE* 17, 173–182 (1991)
- [18] Littlewood, B., Popov, P., et al.: Modelling the effects of combining diverse software fault removal techniques. *IEEE TSE SE-26*(12), 1157–1167 (2000)
- [19] Bloomfield, R., Guerra, S.: Process modelling to support dependability arguments. In: *DSN 2002, International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, D.C (2002)
- [20] Littlewood, B., Wright, D.: Reliability prediction of a software product using testing data from other products or execution environments. *DeVa Project Technical Report 10*, City University London (1996)

Appendix A: General Lemma

If the prior probability density function of the *pdf* of a system is a mixture of probability density functions f_{Q_i} , then substituting any subset of these component distributions with a set of single-point probability masses, one for each of the f_{Q_i} thus substituted, will yield a pessimistic prediction of posterior reliability after observing failure-free demands.

Formally: let random variable Q have probability density function (*pdf*)

$$f_Q(q) = \sum_{i=1}^n p_i f_{Q_i}(q) dq, \tag{7}$$

where $\int_0^1 f_{Q_i}(q) dq = 1$; $p_i > 0, i = 1..n$; and $\sum_{i=1}^n p_i = 1$.

Then, for any two natural numbers t_{past} and t_{fut} ,

$$\frac{\sum_{i=1}^n p_i \int_0^1 (1-q)^{t_{past}+t_{fut}} f_{Q_i}(q) dq}{\sum_{i=1}^n p_i \int_0^1 (1-q)^{t_{past}} f_{Q_i}(q) dq} \geq \frac{p_1(1-q_1)^{t_{past}+t_{fut}} + \sum_{i=2}^n p_i \int_0^1 (1-q)^{t_{past}+t_{fut}} f_{Q_i}(q) dq}{p_1(1-q_1)^{t_{past}} + \sum_{i=2}^n p_i \int_0^1 (1-q)^{t_{past}} f_{Q_i}(q) dq}, \tag{8}$$

(where without loss of generality we have substituted the single component distribution f_{Q_1} ; the l.h.s. of (8) is the posterior reliability), the value of q_1 is

$$q_1 = 1 - \left(\int_0^1 (1-q)^{t_{past}} f_{Q_1}(q) dq \right)^{\frac{1}{t_{past}}}, \quad i = 1..n, \tag{9}$$

and the bound (8) is attained if $f_{Q_1}(q) = \delta(q - q_1)$, where $\delta(x)$ is Dirac's delta function.

Proof

By Hölder's inequality,

$$\int_0^1 (1-q)^{t_{past}} f_{Q_i}(q) dq \leq \left(\int_0^1 (1-q)^{t_{past}+t_{fut}} f_{Q_i}(q) dq \right)^{\frac{t_{past}}{t_{past}+t_{fut}}},$$

i.e.

$$\int_0^1 (1-q)^{t_{past}+t_{fut}} f_{Q_i}(q) dq \geq \left(\int_0^1 (1-q)^{t_{past}} f_{Q_i}(q) dq \right)^{\frac{t_{past}+t_{fut}}{t_{past}}} = (1-q_i)^{t_{past}+t_{fut}}, \tag{10}$$

and (10), together with substituting (9) in (8), proves the lemma.

QED.

Does Software Have to Be Ultra Reliable in Safety Critical Systems?

Peter Bishop^{1,2}

¹ Centre for Software Reliability, City University, London, EC1V 0HB, UK
pgb@csr.city.ac.uk

² Adelard LLP, London, Exmouth House, London., EC1R 0JH, UK
pgb@adelard.com

Abstract. It is difficult to demonstrate that safety-critical software is completely free of dangerous faults. Prior testing can be used to demonstrate that the unsafe failure rate is below some bound, but in practice, the bound is not low enough to demonstrate the level of safety performance required for critical software-based systems like avionics. This paper argues higher levels of safety performance can be claimed by taking account of: 1) external mitigation to prevent an accident; 2) the fact that software is corrected once failures are detected in operation. A model based on these concepts is developed to derive an upper bound on the number of expected failures and accidents under different assumptions about fault fixing, diagnosis, repair and accident mitigation. A numerical example is used to illustrate the approach. The implications and potential applications of the theory are discussed.

Keywords: safety, software defects, software reliability, fault tolerance, fault correction.

1 Introduction

It is difficult to show that software has an acceptably low dangerous failure rate for a safety-critical system. The work of Butler and Finelli [4] and Littlewood and Strigini [21] suggests that there is a limit on the rate that can be demonstrated by testing. Given the difficulty of testing the software under realistic conditions, it is often claimed that this limit is likely to be around 10^{-4} to 10^{-5} failures per hour [18]. As these tests would typically be completed without any failures, we do not know what proportion of the failures are likely to be dangerous, so we have to use the bound derived from testing as the upper bound for the dangerous failure rate as well.

The safety requirement for a software-based system can be far more stringent than the bound established by testing the software, e.g. a target of 10^{-9} per hour for catastrophic failures is required for an individual avionics function [12, 13]. The magnitude of the gap between the demonstrable failure rate and such targets can be illustrated in the following example. With a demonstrable catastrophic failure rate of 10^{-4} /hr per system, 100 critical systems per aircraft and 5×10^7 flight hours per year for civil jet airliners world-wide, around 500,000 fatal aircraft accidents could occur

every year. In practice, statistics on aircraft accidents [2] show that there are around 40 airliner accidents per year worldwide from all causes (including pilot error).

It is clear that real avionics systems perform far better than we are entitled to expect based on testing alone [25], but the actual performance can only be determined *after* the system has been deployed. What we need is a credible argument that would convince a regulator that a software-based system is suitable for use in a safety-critical context *before* it is deployed in actual operation [11].

One alternative to empirical test evidence is the claim that compliance to established standards for safety-critical software will result in software with a tolerable dangerous failure rate. For example, compliance to IEC 61508 Level 4 is linked with dangerous system failure rates as low as 10^{-9} /hr [9]. Unfortunately there is very little empirical evidence to support such a claim.

More credibly, it may be possible to support a claim of perfection if the software is proved correct using formal methods [8]. In this case any failure rate target, even a stringent target like 10^{-9} per hour, would be achievable and the Probabilistic Safety Assessment (PSA) of the overall system could assume the software had a dangerous failure rate of zero. In practice however, few systems have been constructed using formal proof methods, and even these systems cannot be guaranteed to be fault free (e.g. due to errors in requirements or faults in the proof tools [6, 20]).

Another alternative is a risk-informed based design approach [19] which focuses on reducing dangerous failure modes rather than seeking software perfection. Potentially hazardous failure modes are identified and safeguards against these failures are included in the design. However there is no guarantee that a hazard-based approach will identify *all* potential hazards in the real-world environment, and a convincing safety argument would need to show that the hazard identification is complete.

Safety assurance can also be achieved by the use of fault tolerance techniques [1], [14] like design diversity [22] that mitigates failures from individual software components. Software design diversity can reduce the dangerous failure rate of the composite system as the same failure has to occur in more than one software component before it becomes dangerous. These techniques have been used in a range of safety-critical systems [3, 15].

It should be noted that all these strategies for producing safe software are vulnerable to an error in the original specification, i.e. when there is a mismatch between the software requirement and the real world need. This unfortunately also limits the potential for accelerated testing of software against the requirement to reduce the dangerous failure rate bound as the tests will omit the same key features of real-world behaviour.

In practice, systems designers use the defence-in-depth principle to mitigate the impact of dangerous failures in subsystems [7, 10, 20], for example,

- A nuclear protection system failure is covered by an independently designed secondary protection system, manual shutdown and post incident control measures.
- A flight control system failure is covered by diverse flight controls and pilot intervention.

As a result of these strategies, the dangerous failure rate of the system *function* can be lower than that of any individual software component. This can be formalized as:

$$\lambda_{acc} = p_{acc} \lambda \quad (1)$$

where λ_{acc} is the accident rate, p_{acc} is the probability that a dangerous subsystem failure will cause an accident and λ is the bound on the dangerous software failure rate established by testing.

The problem lies in achieving and justifying an appropriate value of p_{acc} . Empirical studies of software fault tolerance techniques like design diversity [16, 26] suggest that reductions of no more than two orders of magnitude can be achieved, while human intervention under stress may only result in another order of magnitude reduction in the dangerous failure giving a total reduction of 10^{-3} . If a dangerous failure rate of 10^{-4} /hr can be demonstrated from actual flight testing, it might be argued that the accident rate due to failure of the avionics subsystem is no worse than 10^{-7} /hr, but this could still be insufficient to demonstrate the required target (e.g. 10^{-9} /hr for an avionics function).

In this paper we will present a probabilistic software failure model that can be used to claim a lower contribution to the accident rate from dangerous software faults. This approach is novel and potentially controversial as it requires certification bodies to accept an argument based on a low average risk over the system lifetime, but with the possibility of a higher instantaneous risk when the system is first introduced.

2 Basic Concept

Software failures need to be handled in a different way to hardware failures because a systematic software defect can be fixed—once we know what the problem is, it can be removed. In the best case, each software fault need only fail once if it is successfully fixed in all instances immediately after failure, so we consider that it is inappropriate to use a fixed failure rate for software in a safety justification. We also need to take account of the fact that software failures need not be catastrophic (i.e. cause an accident), because there can be mitigations outside the software-based component.

In the most basic representation of this idea, we consider the failures caused by a single fault in the software (the impact of multiple faults will be considered later).

Clearly the number of failures that occur before the fault is successfully fixed depends on the probability of diagnosing a fault and then repairing it correctly [24]. In the basic model, we make the following assumptions.

- The conditional probability that a fault is diagnosed when a software failure occurs is p_{diag} .
- The conditional probability that a fault is repaired correctly after diagnosis is p_{repair} .
- The success of diagnosis and repair is independent of the number of previous failures.
- No further failures can occur *in any software instance* until the fix attempt has finished.

Given the independence assumptions made on diagnosis and repair, the probability of a fault being successfully fixed after each failure is:

$$P_{fix} = P_{diag} P_{repair} \tag{2}$$

Given the assumption that no failures can occur during a fix, a simple transition model can be used to model the fixing process as illustrated in Fig 1.

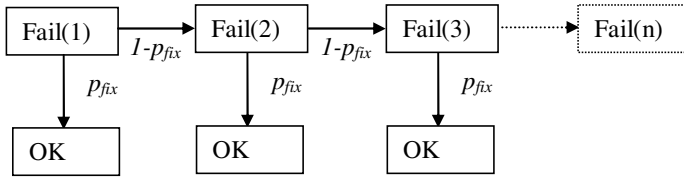


Fig. 1. Fault correction model

So for $p_{fix} = 0.5$, there is a 50% chance of removing a fault after the first failure; 25% after the second failure; and so on. The mean length of this failure sequence, n_{fail} , is:

$$n_{fail} = \sum_{n=1, \infty} n \cdot p_{fix} (1 - p_{fix})^{n-1} \tag{3}$$

Using a standard result for this geometric series [27], this reduces to:

$$n_{fail} = \frac{1}{P_{fix}} \tag{4}$$

This represents the expected number of failures over an infinite period of time, caused by a single software fault operating within the whole fleet of software-based units. If there are N faults that cause dangerous failures, then the expected number of fleet failures due to these faults is bounded by:

$$n_{fail} \leq \frac{N}{P_{fix}} \tag{5}$$

The other element of the model is based on the fact that safety-related computer-based systems typically operate within a fault-tolerant architecture (as discussed earlier). We can represent this external mitigation of a dangerous software failure by the probability p_{acc} that an accident occurs after a dangerous software failure.

It follows that the expected number of accidents over the lifetime of the fleet due to N dangerous faults is bounded by:

$$n_{acc} \leq \frac{N p_{acc}}{P_{fix}} \tag{6}$$

This equation implies that if $N \ll p_{fix}/p_{acc}$, the expected number of accidents $n_{acc} \ll 1$. This represents the case where there is a high probability that all N dangerous faults will be diagnosed and removed before the first accident occurs. A value of n_{acc} well below unity is effectively equivalent to the probability of an accident over the lifetime of the fleet due to failures of the software component.

Given the assumption that no further software failures occur during a fix attempt, the failure rate of the software has no impact on the maximum number of accidents in the fleet. This assumption could be satisfied if all failures resulted in an instantaneous fix attempt, or more realistically, it could be met if the fleet was grounded immediately after failure while the fix attempt is made.

This independence between the upper bound on failure rate and the number of accidents is particularly useful in cases where the failure rate bound has not been estimated correctly, e.g. due to a flaw in the specification. Such a flaw would invalidate any failure rate estimate based on testing, but the accident bound derived from equation (6) would still be valid provided N included an estimate for dangerous specification flaws. This differs from hardware where the instantaneous failure rate is often assumed to be constant, so expected accidents always increase with fleet usage.

3 Impact of Delayed Fixing

The basic model makes a strong assumption that no further failures will occur after a dangerous failure is observed. In many cases however, the fleet containing the software components will continue to operate after the failure has occurred. Clearly, if repair is delayed, further failures could occur within the fault fixing interval.

Initially we will consider the case of a single fault (extension to N faults will be addressed later). Let us define:

λ	as the upper bound on the software failure rate
Δt_{fix}	as the time needed to perform diagnosis and repair
$\tau(t)$	as the total execution time of the software fleet at elapsed time t

We further assume that:

- No new faults are introduced when the software is fixed.
- The failure rate bound λ is unchanged if the fix attempt is not successful.

These assumptions are also quite strong. New faults are known to occur occasionally but if the new fault is in the same defective code section, it can be modeled as the *same* fault with a reduced p_{fix} value.

The second assumption is conservative if the rate actually decreases after a repair (e.g. due to a partial fix). An increase in failure rate would not be conservative, but it might be argued that the rate is bounded by the execution probability of the faulty code section. The assumption that the failure rate is unchanged by unsuccessful fixes makes this process mathematically equivalent to fixing a fault with failure rate λ with probability p_{fix} at a time Δt_{fix} after a failure was observed.

To estimate the impact of delayed fixing, we first define the time needed before an average length failure sequence terminates, t_{fail} , as the value that satisfies the equation:

$$\lambda \tau(t_{fail}) = n_{fail} \quad (7)$$

From equation (4), this is equivalent to:

$$\lambda \tau(t_{fail}) = \frac{1}{P_{fix}} \quad (8)$$

It can be shown using Jensen's Inequality [17] that, if the execution time function $\tau(t)$ is convex (i.e. the gradient is constant or increases over calendar time), the total number of failures, n_{fixed} , when the fix delay is included is bounded by:

$$n_{fixed} \leq \lambda \tau(t_{fail} + \Delta t_{fix}) \quad (9)$$

We now consider a situation where there are N dangerous faults. Most reliability models assume the failures of the individual faults occur independently. If this assumption is made, the failure rates sum to λ , but we will take a worst case scenario where:

- The failure rate of each fault is λ .
- Failures occur simultaneously for N faults.
- Only one fault can be fixed after a failure.

In this worst case scenario there will N times more failures than a single fault and the failures will occur at the same frequency, λ , as the single fault case. This failure sequence is equivalent to having a single fault with a p_{fix} probability that is N times worse, i.e. where:

$$p'_{fix} = \frac{p_{fix}}{N} \quad (10)$$

The bound in equation (9) can therefore be generalised to N faults as:

$$n_{fixed} \leq \lambda \tau(t'_{fail} + \Delta t_{fix}) \quad (11)$$

where:

$$\lambda \tau(t'_{fail}) = \frac{1}{p'_{fix}} \quad (12)$$

So from equation (10), t'_{fail} has to satisfy the relation:

$$\lambda\tau(t'_{fail}) = \frac{N}{p_{fix}} \tag{13}$$

It follows that the worst scale factor k due to delayed fixing, is:

$$k = \frac{n_{fixed}}{n_{fail}} \leq \frac{\tau(t'_{fail} + \Delta t_{fix})}{\tau(t'_{fail})} \tag{14}$$

The basic principle for scaling the bound is illustrated in Fig 2. Without fixing, the expected number of failures increases to infinity. With fixing and no delay, the number cannot exceed the basic bound, and would take a time t'_{fail} for the bounding number of failures to occur. With a fix delay, the bound is increased to allow for the additional failures that can occur in the extra time Δt_{fix} needed for fault repair.

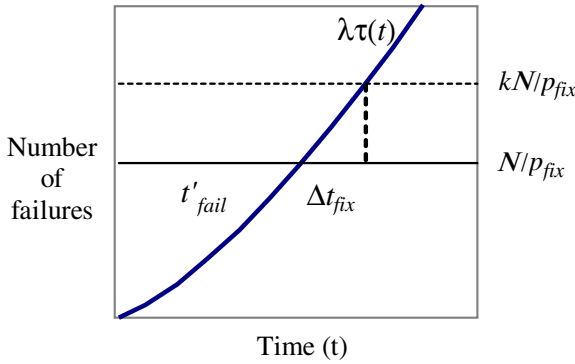


Fig. 2. Effect of fix delay on the expected number of failures

We can also compare the expected number of failures with fault fixing (n_{fixed}) against the expected number without fault fixing, ($n_{unfixed}$), namely:

$$n_{unfixed} = \lambda\tau(t_{fleet}) \tag{15}$$

where t_{fleet} is the calendar time that the software is in operation in the fleet. So the failure reduction (and hence accident reduction), r , achieved by fault fixing is:

$$r = \frac{n_{fixed}}{n_{unfixed}} \leq \frac{\tau(t'_{fail} + \Delta t_{fix})}{\tau(t_{fleet})}, \quad t'_{fail} + \Delta t_{fix} < t_{fleet} \tag{16}$$

Application to Demand-based Systems. Equation (6) is directly applicable to any demand-based system if the fleet is grounded during a fix attempt. The increase k caused by delayed fixing can be calculated for demand-based systems using equations (13) and (14) provided we can derive an equivalent failure rate bound λ .

For example, if we know there is an average demand rate of d demands per unit of execution time and the upper bound on the probability of failure on demand is f (e.g. from accelerated testing), the effective failure rate per unit of execution time is:

$$\lambda = fd \quad (17)$$

4 Theory Applied to an Avionics Example

To illustrate the potential gains achievable by including fault fixing, we will apply the theory to a hypothetical avionics example with the following parameters

c	10 unit sales per month
u	0.6 (fraction of time in use)
N	1
λ	10^{-4} failures/ hour
p_{fix}	0.1
p_{acc}	0.001

Note that the figures used are only estimates, but are considered to be realistic. The number of dangerous faults N is taken to be one as we assume thorough levels of testing and analysis (especially for the safe-critical portions of the software). The assumed failure rate represents a year of realistic flight testing (e.g. in ground based tests and actual test flights). The probability of an accident p_{acc} is assumed to be small because an aircraft is engineered to tolerate failures of specific components (via standby systems, pilot override, etc). The p_{fix} probability is actually the product of diagnosis and repair probabilities, i.e. $p_{fix} = p_{diag} p_{repair}$. For critical software we expect the repair probability achieved by the software support team to be close to unity, so p_{fix} is largely determined by the diagnosis probability, which is estimated to be around 0.1 as any hazardous incidents will occur in-flight, and diagnosis relies on later reconstruction of events based on in-flight recording data.

If the fleet is grounded after a dangerous failure, the basic model applies and we would expect 10 failures (from equation (3)) and 0.01 accidents (from equation (6)) over the fleet lifetime.

If there is delayed fixing, the k value has to be computed using the execution time equation. With a linear growth in the fleet of avionics units at c per month the execution time function can be shown to be:

$$\tau(t) = \frac{cu}{2} t^2 \quad (18)$$

We can use equations (13) and (14) and the execution time function (18) to compute the scale-up factor k . The impact of different fix delay times (Δt_{fix}) on k and the expected number of accidents is shown in Table 1.

Table 1. Expected Accidents Over Infinite Time for Different Software Fix Times

Δt_{fix} (months)	k	n_{fixed}	n_{acc}
0	1	10	0.010
1	1.3	13	0.013
2	1.7	17	0.017
3	2.1	21	0.021

With a 3 month delay in fixing, the bound on the expected number of fleet accidents is only double the number predicted by the basic model (with no fix delay).

The upper bound on the mean accident rate λ_{acc} over the fleet lifetime is:

$$\lambda_{acc} \leq \frac{n_{acc}}{T_{fleet}} \tag{19}$$

where T_{fleet} is the total execution time of all the avionics units.

From equation (18), if unit sales continued for 5 years, the total number of operating hours, T_{fleet} , is around 1.6×10^7 hours. So the upper bound on the mean accident rate λ_{acc} over the fleet lifetime for a 3 month fix delay is:

$$\lambda_{acc} \leq 1.3 \times 10^{-9} \text{ accidents per hour}$$

This bound on the mean accident rate is close to the target of 10^{-9} accidents per hour required in avionics standards [12][13]. The bound could be reduced to less than 10^{-9} accidents per hour if a shorter fix delay is used (e.g. 1 month).

By comparison, if we only relied on external accident mitigation, the bound on the mean accident rate would be the same as the initial rate $p_{acc}\lambda$. For the avionics example, the expected rate would be 1×10^{-7} accidents per hour.

So for this choice of model parameters, the inclusion of a fault removal model has reduced the expected accident rate over the fleet lifetime by two orders of magnitude. Clearly the reduction varies with the parameters used. Table 2 shows the accident reduction r achieved by fault fixing for different failure rates assuming a 3 month delay in fixing and a 5 year operating period.

Table 2. Accident Reduction for Different Software Failure Rates

λ (per hr)	Mean Accidents / hr		Reduction factor r
	(no fix)	(fix)	
10^{-3}	10^{-6}	3.5×10^{-9}	0.0035
10^{-4}	10^{-7}	1.3×10^{-9}	0.013
10^{-5}	10^{-8}	0.8×10^{-9}	0.08

It is apparent that the greatest reduction occurs when the software failure rate λ is high. This is not surprising as the mean accident rate is relatively stable (regardless of λ) when there is fault fixing, but it increases linearly with λ without fault fixing.

5 Discussion

The fault fixing model predicts an upper bound on the total number of software failures (and associated accidents) over the fleet lifetime. The impact of fault fixing is greatest in large fleets where the expected number of failures without fixing would greatly exceed the expected number with fault fixing. In the avionics example, the claimed accident rate can be two orders of magnitude less than would be predicted from testing alone. In particular, once a relatively modest (and demonstrable) level of software reliability is achieved, further reductions in failure rate make little difference to the ultimate number of failures and accidents.

This type of probabilistic argument is not currently accepted in safety standards or by certification and regulatory bodies. Early users of the system could be placed at greater risk if the instantaneous failure rate is close to the limit established by testing. However it might be more acceptable as a *support* to a primary argument such as a claim of zero faults in critical portions of the software. The supporting argument would be that, even if the claim of zero dangerous faults is invalid, there is high probability that a software fault never causes any accident over the lifetime of the fleet (e.g. 98% in our avionics example).

If the theory is valid, equation (6) can also be helpful in choosing design trade offs. We note that an order of magnitude change in the predicted number of accidents can be achieved by an order of magnitude change in either: N , p_{diag} , or p_{acc} . Knowing the contribution of these parameters, design trade-offs can be based on cost and technical feasibility. For example, sending extra data to a shared black-box data recorder to improve p_{diag} might be more cost effective than additional effort to reduce N . Alternatively installing a backup system using different technology might double the cost but improve p_{acc} by orders of magnitude.

The theory also shows that the operational context can affect the accident probability. Obviously the repair probability p_{diag} directly affects the number of accidents, and we can minimise the scale-up k due to delayed fixing by considering equations (13) and (14). For example k might be reduced by decreasing the fix delay time Δt_{fix} or by reducing the growth in usage $\tau(t)$ for some trial period.

To successfully apply the model, evidence will be needed to show that the model parameter estimates are either realistic or at least conservative. Values, like N , could be derived from past experience with similar systems, (e.g. analysing FAA directives [5, 25]) but further research is needed on quantifying the model parameters.

More generally, the same theory should be applicable to *any* systematic design fault that is amenable to fault fixing such as software security vulnerabilities, hardware design faults or requirements faults.

6 Summary and Conclusions

This paper has presented a basic fault-fixing model that shows there is an upper bound on the expected number of dangerous software failures if faults are diagnosed and fixed. If the fault fixing is immediate, this bound is independent of the software

failure rate. When this bound on failures is combined with external failure mitigation, there can be a high probability that an accident is *never* caused by dangerous software failure regardless of the size of the fleet using the software-based component.

We have also presented a refinement of the basic model that allows the bound to be increased to allow for a delay in fixing a detected fault. This revised bound is dependent on the software failure rate, but the increase is typically quite small.

The theory was illustrated by an aircraft avionics example where fault fixing reduced the expected number of accidents by around two orders of magnitude over the fleet lifetime.

If the assumptions behind the theory are valid, it could provide an additional means of arguing that critical software-based systems are safe prior to deployment even though ultra high reliability of the software cannot be demonstrated by prior testing.

The theory might also be helpful in making design and support trade-offs to minimize the probability of an accident.

We also suggest that the theory could be applicable to *any* systematic fault (e.g. in requirements, hardware, software or mechanical components).

Further empirical research is recommended to validate the model assumptions and quantify the model parameters.

Acknowledgments. The author wishes to thank Bev Littlewood, Lorenzo Strigini, Andrey Povyakalo and David Wright at the Centre for Software Reliability for their constructive comments in the preparation of this paper.

References

1. 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(1), 11–33 (2004)
2. Boeing, Statistical Summary of Commercial Jet Airplane Accidents Worldwide Operations 1959 – 2010. Aviation Safety, Boeing Commercial Airplanes, Seattle, Washington, U.S.A (June 2011)
3. Briere, D., Traverse, P.: Airbus A320/A330/A340 electrical flight controls — a family of fault-tolerant systems. In: Proc. 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), pp. 616–623 (June 1993)
4. Butler, R.W., Finelli, G.B.: The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering* 19(1), 3–12 (1993)
5. FAA Airworthiness Directive database, http://rgl.faa.gov/Regulatory_and_Guidance_Library/rgAD.nsf/MainFrame
6. Graydon, P.J., Knight, J.C., Yin, X.: Practical Limits On Software Dependability: A Case Study. In: Real, J., Vardanega, T. (eds.) *Ada-Europe 2010*. LNCS, vol. 6106, pp. 83–96. Springer, Heidelberg (2010)
7. Hecht, H., Hecht, M.: Software reliability in the system context. *IEEE Transactions on Software Engineering* 12, 51–58 (1986)
8. Hinchey, M.G., Bowen, J.P.: *Industrial-strength formal methods in practice*. Springer (1999)
9. IEC, Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508 ed. 2.0, International Electrotechnical Commission (2010)

10. INSAG, Defence in Depth in Nuclear Safety. INSAG 10, International Nuclear Safety Advisory Group (1996)
11. Jackson, D., Thomas, M.: Software for dependable systems: sufficient evidence? National Research Council (U.S.). National Academic Press (2007) ISBN 978-0-309-10394-7
12. Joint Airworthiness Authority, Joint Airworthiness Requirements, Part 25: Large Aeroplanes JAR 25 (1990)
13. Joint Airworthiness Authority, Advisory Material Joint (AMJ) relating to JAR 25.1309: System Design and Analysis. AMJ 25.1309 (1990)
14. Kanoun, K., Laprie, J.-C.: Dependability modeling and evaluation of software fault-tolerant systems. *IEEE Transactions on Computers* 39(4), 504–513 (1990)
15. Kantz, H., Koza, C.: The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity. In: 25th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-25), pp. 453–458 (June 1995)
16. Knight, J.C., Leveson, N.G.: An empirical study of failure probabilities in multi-version software. In: Digest. FTCS-16: Sixteenth Annual Int. Symp. Fault-Tolerant Computing, pp. 165–170 (July 1986)
17. Krantz, S.G.: Jensen’s Inequality. In: Section 9.1.3 in *Handbook of Complex Variables*, p. 118. Birkhäuser, Boston (1999)
18. Laprie, J.-C.: For a product-in-a-process approach to software reliability evaluation. In: Third International Symposium on Software Reliability Engineering (ISSRE 1992), pp. 134–139 (October 1992)
19. Leveson, N.G.: *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press (2012) ISBN 978-0262016629
20. Littlewood, B., Rushby, J.: Reasoning about the Reliability of Diverse Two-Channel Systems in Which One Channel Is “Possibly Perfect”. *IEEE Transactions on Software Engineering* 38(5), 1178–1194 (2012)
21. Littlewood, B., Strigini, L.: Validation of Ultra-High Dependability for Software-based Systems. *Communications of the ACM* 36(11), 69–80 (1993)
22. Lyu, M.T.: *Software Fault Tolerance*. John Wiley & Sons, Inc., New York (1995) ISBN:0471950688
23. Radio Technical Commission for Aeronautics, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178C. RTCA, Washington, DC (December 2011)
24. Smidts, C.: A stochastic model of human errors in software development: impact of repair times. In: *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE 1999)*, pp. 94–103 (1999)
25. Shooman, M.: Avionics Software Problem Occurrence Rates. In: *Seventh International Symposium on Software Reliability Engineering (ISSRE 1996)*, pp. 55–64 (1996)
26. van der Meulen, M.J.P., Revilla, M.A.: The Effectiveness of Software Diversity in a Large Population of Programs. *IEEE Transactions on Software Engineering* 34(6), 753–764 (2008)
27. Zwillinger, D.: *CRC Standard Mathematical Tables and Formulae*, 31th edn., pp. 630–631. CRC Press, Boca Raton (2003)

The SafeCap Platform for Modelling Railway Safety and Capacity

Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky

School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
{alexei.iliasov,ilya.lopatkin,alexander.romanovsky}@ncl.ac.uk

Abstract. This paper describes a tooling platform that supports reasoning about railway capacity while ensuring system safety. It uses a Domain Specific Language (DSL) that allows signalling engineers to design stations and junctions, to check their safety and to evaluate the potential improvements of capacity while applying various alteration patterns that change the railway schemas. The platform uses a combination of model checking and SMT solving to verify system safety in the most efficient and user-friendly way. It includes several plug-ins that evaluate various capacity parameters. The tool uses the Eclipse technology, including its EMF and GMF frameworks. It has been developed in close cooperation with the Invensys Rail engineers and applied in a variety of medium-scale projects, which has demonstrated its ability to help understand the effects that changes in the plans and schemas can potentially have on capacity.

Keywords: Domain-specific language, Eclipse, EMF, GMF, Event-B, ProB, SMT solvers, model transformation, capacity-improving patterns.

1 Introduction

Ensuring railway safety has been an area of successful application of formal methods and tools (the most famous example being the use of B by Matra for developing the automated metro line 14 in Paris [1]). There is a substantial body of work on formal verification of railway safety. For example, paper [2] defines a model-based development method for railway control systems that uses a domain-specific notation from which the models of the controllers and the domain are generated to be used to verify the safety properties by model checking. Another domain-specific language (DSL), called the Train Control Language (TCL), is in the core of the method in [3], used to verify and validate the safety of stations; this is achieved by transforming the TCL models into the Alloy models used for constraint solving. These two methods are supported with tools, while the models are created in DSLs and automatically mapped into a formal notation used for safety verification.

There are many reasons why improving railway capacity is now one of the top priorities for the railway industry, including the need to satisfy passengers' requirements, tackle global warming and save running costs. Clearly, any improvements in capacity must not undermine system safety. Except for the work conducted by our collaborators in the SafeCap project [4, 5], the existing work on formal verification of railway safety does not address safety and capacity in an integrated way.

2 The SafeCap Approach

Designing railway nodes (junctions and stations) for high capacity is an art: experienced engineers know the patterns that lead to success or failure and are aware of the bottlenecks that exist within their designs. However, such an intuitive approach lacks scientific foundations, which are the basis of any advanced engineering practice. To this end, the SafeCap project [5] aims to provide a scientifically sound framework that will allow railway engineers to study railway nodes as well as design patterns, addressing safety and capacity in an integrated way

The general approach in SafeCap consists of the following steps. First, engineers model a junction or a station and evaluate its capacity. At the second step, they try to improve capacity by applying alteration patterns (in effect model transformations). These patterns can capture various changes in the route design, track layout and signalling, etc. At the next step the capacity of the modified model is evaluated. The safety of models is formally proven at every step. This approach allows the engineer to experiment with various designs to achieve better capacity.

The importance of tool-supported formal reasoning is well recognised by both engineers and academics. This is why in the SafeCap project we have been developing a tooling platform that can support domain experts, including signalling engineers, in making rigorous decisions about improving capacity. To make our tool more acceptable for industrial users, we follow the push-button approach, in which safety verification and capacity evaluation are conducted in a transparent fashion. Another choice we made in designing the platform is the exclusive use of a graphical and intuitive DSL, saving engineers from the need to understand intricate formal notations. One more feature of the tool is extensive support for representing and reusing modelling patterns that capture best practice and experience. All this should help the platform to be widely accepted by industry.

3 Domain Specific Language

SafeCap offers a fairly compact core DSL. The basic element of a SafeCap schema is the definition of railway topology. The main concepts of the DSL are tracks, nodes, ambits (train detection units), routes, lines and rules. The SafeCap DSL is a *formal* language: a schema is interpreted as a hybrid transition model - a model mixing continuous and discrete behaviours. The discrete part is employed to derive static verification conditions (theorems) and, as a supplementary technique, to help discover transition traces leading to the violation of safety conditions. The continuous part refines the discrete part with the notions of train acceleration/deceleration, point switching and driver reaction times, and so on.

Static verification conditions are logical constraints over the elements of a schema expressing requirements to schema topology, formation of routes, placement of speed limits and signalling rules of routes and points. If all such conditions are discharged, it is guaranteed that the schema is safe for any possible rail traffic. Together, the conditions form the *theory of the SafeCap schemas* [6].

4 The SafeCap Tooling Platform

The following aims were set for building the tool support. Our plan was to use an industry-strength modern technology that would allow us to create a tooling environment that is open for extensions and supports graphical modelling using our DSL. The technology should have mature support for model transformation, to allow us to deal with different representations of the schemas. We wanted to use a technology that smoothly supports tool development on various operating systems.

We have decided to use Eclipse as the basis for our tool. Eclipse is a mature and extensible IDE framework that offers a powerful and flexible customisation framework. The concept of proving extra functionality via self-contained plug-in projects makes it easier to support and maintain a large project with a team of developers.

The SafeCap platform uses the Eclipse Modelling Framework (EMF) - a versatile tool for the implementation of a custom domain-specific language. We have found it sufficient to capture the static part of SafeCap DSL.

We use the Eclipse Graphical Modelling Framework (GMF) that builds on top of the EMF to provide means for the rapid construction of graphical editors manipulating EMF models. The GMF offers a layer of abstraction above the code level to define core properties of editors. This is how the SafeCap editor is built.

Openness is the key property of the platform. To make the tool truly open and customisable, it was decided to support a scripting language that executes directly within the platform and is used to implement its main functionalities (verification, capacity assessment and improvement patterns). This frees users from having to learn Java and Eclipse API in order to customise the platform logic. The Epsilon family of languages [7] was deemed perfectly suitable for the task. Among other things, the SafeCap platform uses Epsilon to support creation and application of user-defined patterns for transformation of schemas. At the moment, these are used during modelling and capacity improvement. We plan to extend the application of patterns to support automatic search for transformations that improve capacity.

5 Safety Verification Architecture

Safety is verified by formulating railway schema properties and operational principles in a rigorous notation. The desire to have a strict mathematical foundation spanning the principal aspects of railway operation has led to a distinctive four-layered architecture of the verification back-end.

In this architecture, *the first (schema topology theory) layer* is responsible for verifying logical conditions expressed over schema physical topology (i.e., track connections, point placement) and logical topology (i.e., routes and lines as paths through a schema). These verification conditions include connectivity of track topology (no isolated pieces of track), continuity of routes and lines, etc. Typically these conditions do not uncover problems with an existing track layout, for any such defect would have a profound effect on the overall integrity - something unlikely to not have been discovered in an operational railway. It is the automated and semi-automated alteration and generation of track layouts (e.g., via the improvement patterns) that necessitates a careful inspection of these basic properties.

In *the control table theory layer* the conditions for operational safety are defined. These are derived, via formal proof, from a set of discrete (inertia-less) train movement rules into a set of theorems over the properties of control tables. We depart from the convention of associating control rules with trackside signals. Instead, we consider a more general situation where differing set of signalling rules are applied depending upon the ultimate train destination or train type. The conditions proven for a control table demonstrate such properties as the absence of potential collision (as may happen, for instance, when a proceed aspect is given while a protected part of track is still occupied) and derailment (due to incorrect point setting or point movement under a train). Certain properties, notably the control of the approach speed by means of the timed occupation of a track section, are not verified at this stage, as formalisation at this layer does not capture train inertia. Speed limit conformance and other time-related properties are formulated at the fourth layer.

The first two formalisation layers do not define the notion of a train. However, there is a link between the conditions over control tables expressed in the second layer and the notion of train movement. The definition of the latter is the purpose of *the third (discrete driving model) layer*. This layer defines principal events that are observed during railway operation: train movement, route reservation, point locking, route cancellation and so on. On the basis of these one can state operational safety principles by using safety invariants (e.g., trains are not overlapping) or by explicitly modelling possible operation faults, e.g., uncontrolled carriage movement on an adjacent route (to ensure the correctness of flank protection). Apart from its role in the validation of the first two layers, the discrete operational rules of the third layer are used to visually animate train movements over a given schema. There are two main applications for such an animation: replaying the results of model checking of discrete driving rules in order to pinpoint the source of an error in a topology or a control table; and helping an engineer to understand how trains may travel through a schema with a given set of control rules.

Train inertia is the focus of *the final modelling (inertial driving model) layer*. This layer is built from the same primitives as the rules of the third layer, with the main difference that discrete rules are defined as atomic sequences of primitive steps (for instance, a train head movement, in one step, moves the train head position, unlocks an ambit and, sometimes, resets a signal to red), whereas the more detailed inertial model accounts for the duration of every action and executes concurrent actions in a more realistic way than the lock-step fashion of the discrete model. This transition to inertial, timed transformation uncovers a wealth of concerns. The most essential one, perhaps, is whether the properties of operational safety still hold for the inertial model. There is not an unconditional answer to this question. In fact, one can show a certain railway configuration accepted by the inertial model to be unsafe. The ability to run a detailed simulation (with a fixed service pattern) that accurately deals with track gradient, engine performance, train weight, etc., gives access to rich information about realistic node performance. The current version of the tool provides an implementation of the inertial models used is simulation only. The on-going work will support the full approach for analysing both safety and capacity.

The primary mechanism for verifying safety is constraint solving. After the schema is defined, the platform mechanically derives verification conditions and translates them into an Event-B model that serves as an input notation for an SMT-LIB-compliant SMT solver (Yices [8]). One downside of applying constraint solving in our context is that we cannot always receive a useful feedback to indicate the source of a problem should an error be discovered. To compensate for this, whenever a SMT solver detects a problem, the tool runs the ProB model checker [9]. Unlike solvers, the model checker explores the state space of a discrete transition system that gives semantics to the SafeCap schemas represented as the Event-B model with the DSL axioms defined in the machine context. It is thus able to report a sequence of steps (discrete train movements, point switching, etc.) that leads to violation of a safety condition (e.g. a collision or derailment). In most cases, such a sequence can be visually replayed by the tool platform to help the user debug the schema.

This approach to safety verification is superior to the traditional analysis that uses model checkers in terms of efficiency and of the size of the models analysed. Another advantage is that the feedback received by users is expressed in the DSL.

6 Reasoning about Capacity

Depending on the objectives of modelling, the capacity assessment is conducted by calculating a value according to one of the predefined formulae or by running a detailed simulation of train movements. The platform has a range of plug-ins supporting various capacity metrics. Below we introduce some of the capacity plug-ins.

Theoretical line capacity. This criterion assesses the capability of a line to support a certain amount of traffic irrespective of signalling constraints and safety requirements. The plug-in calculates a theoretical line capacity in trains per second.

Critical section. One obvious weakness of the theoretical capacity method is its inability to capture the notion of shared or intersecting track and hence the interference of traffic on crossing lines that leads to decreased capacity. The plug-in calculates the maximum time a train on a given line occupies the critical section.

Wasted track capacity. One could take a different viewpoint and try to assess how efficiently the existing signalling makes use of the available track. Assuming a certain traffic pattern, the plug-in measures the minimum amount of free track observed during a traffic scenario. The fundamental idea is that line interference and inefficient signalling tend to increase train headways.

Cumulative travelled distance. This plug-in measures the total distance travelled by all the trains that have entered the junction. The measurement is done for a set period of time and starts after a certain delay in an attempt mitigate the effect of the initial absence of traffic. The guiding intuition here is that a more efficient layout and signalling favour a balance between higher average speed and less wasted capacity.

Satisfaction of schedule. If there is a detailed specification of a desired traffic pattern through a junction then one may take the ability to satisfy the pattern as a measure of capacity. A service pattern defines train kinds and the times trains appear at

boundary nodes of a schema during a period known as pattern duration. The plug-in checks schedule satisfaction by running a timed simulation of the hybrid model.

7 Experience of Using the Platform

To evaluate the SafeCap platform, we developed two large-scale examples modelling the existing UK stations. The primary objective for modelling the case studies was to improve and evaluate the scalability and usability of the tool. The first case study modelled is a fragment of the Thameslink line around the Kentish Town station. The fragment is 5.5 km long, with the model containing 90 ambits and 63 routes. The modelling took 37 man-hours. Our main activity was the translation of traditional railway diagrams into the SafeCap DSL using the platform.

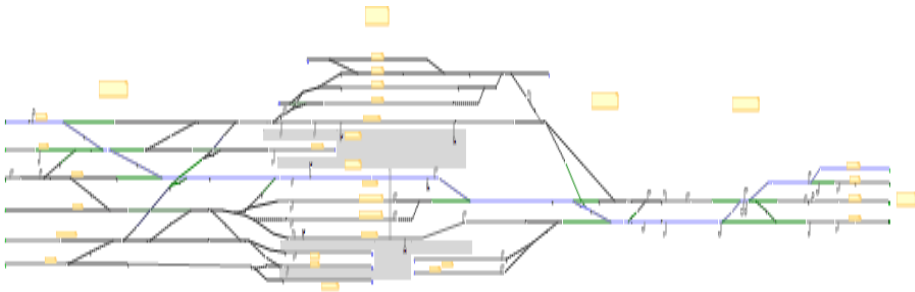


Fig. 1. Carlisle central station and junctions on approaches

The second case study is the Carlisle Citadel station with the North, South, and Caldew junctions (see Fig. 1). The modelled fragment is 2.6 km long and is made of 70 ambits and 79 routes. The translation activity took 45 man-hours. The safety verification of the schema topology requires 35 minutes on a modern computer and goes through 877 individual instantiated conditions. Fig. 2 shows the editing mode of the SafeCap platform for this model. The main view gives a visual representation of the track layout and some signalling mark-up for the Carlisle station platforms.

The platform provides a blocking diagram tool that tries to reduce the time of schedule satisfaction by identifying and reducing the wasted track capacity. This was used to conduct a short capacity-improvement experiment (Fig. 3). The total time during which a set of trains on particular lines (a service pattern) travels through the schema was taken as a measurement of capacity. Using the tool to identify a bottleneck made it possible to improve the capacity by 5%. (We should note here that we could not claim that this will always improve the capacity of the real station, as there are many factors to be considered, such as cost or validity of our assumptions.)

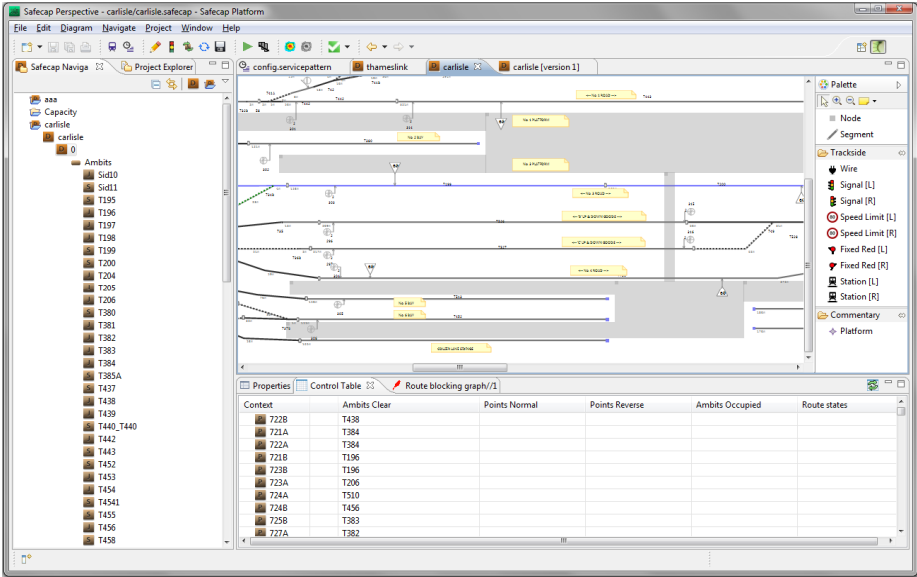


Fig. 2. Model of Carlisle station

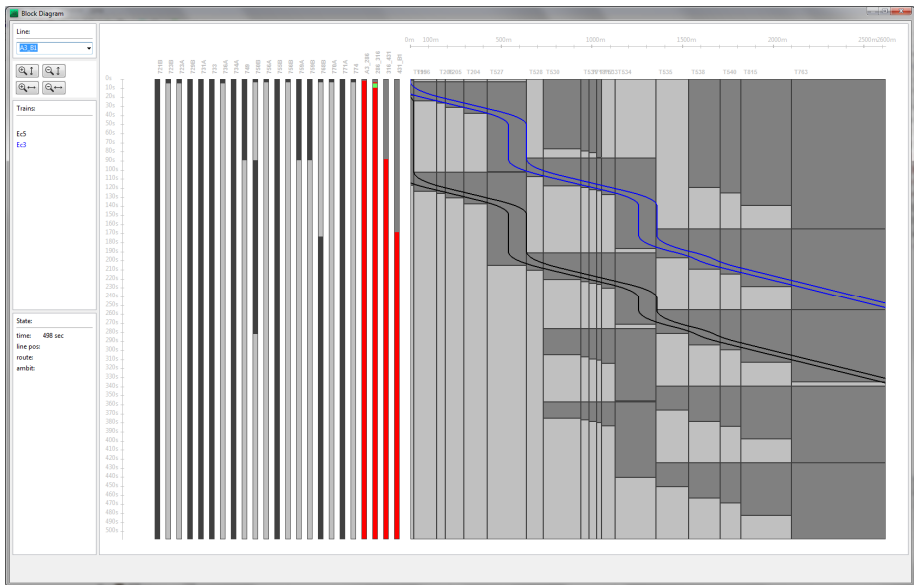


Fig. 3. Blocking diagram

8 Conclusion and Future Work

The SafeCap layered architecture proved to be useful for efficient railway modelling using formal techniques developed by computer scientists. We are now training a

group of engineers to get their feedback, understand how this tooling fits into their current practice and capture their best practice in a set of capacity-improving patterns.

The open nature of the platform will allow us to substantially extend its functionality in the near future. We are already working on adding plug-ins for reasoning about energy and cost. This will be an important step in making the SafeCap approach practical, as capacity improvements should be always evaluated against their cost and energy implications. Another potential extension is to integrate tool support for reasoning about line capacity in Timed CSP [4] to allow us to add extra capabilities of capacity measuring using timed-based formal reasoning.

One of the advantages of applying formal modelling in the railway domain is that it makes it equally easy to verify the existing operational principles and novel, untested ideas. The level of confidence a formal approach brings is especially valuable in overcoming the healthy scepticism towards novelty in the field known for its conservatism. Thus, it perhaps makes sense to make a step further and claim that a set of uniform operational principles demanding the same signalling practice homogeneously deployed across a network is no longer a relevant approach at the age when most railway aspects are controlled by a computer. In-cab signalling and radio-based train position detection, like that of ETCS Level 3, can be used to implement a dynamically reconfigurable signalling logic that adapts, nearly instantaneously, to real-time capacity demands and emerging traffic patterns. Adapting our layered approach to deal with the task of dynamically generating formalisation layers will be an exciting technological challenge.

More information, including demos, examples, videos and documentation, can be found on the platform site (safecap.sourceforge.net). The tool is developed on SourceForge and publicly distributed. A users' and developers' community will be created to ensure the platform quality and to improve its applicability.

Acknowledgement. We are grateful to Simon Chadwick and Dominic Taylor for their comments on the earlier drafts of the paper.

References

1. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: *Météor: A successful application of B in a large project*. In: Wing, J.M., Woodcock, J. (eds.) *FM 1999*. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
2. Haxthausen, A.E., Peleska, J., Kinder, S.: *A formal approach for the construction and verification of railway control systems*. *Formal aspects of computing* 23, 191–219 (2011)
3. Svendsen, A., Moller-Pedersen, B., Haugen, O., Endresen, J., Carlson, E.: *Formalizing train control language: automating analysis of train stations*. *WIT TBE* 114 (2010)
4. Isobe, Y., Moller, F., Nguyen, H.N., Roggenbach, M.: *Safety and Line Capacity in Railways – An Approach in Timed CSP*. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) *IFM 2012*. LNCS, vol. 7321, pp. 54–68. Springer, Heidelberg (2012)
5. *The SafeCap project: Overcoming the railway capacity challenges without undermining rail network safety*, <http://safecap.cs.ncl.ac.uk>
6. Iliassov, A., Romanovsky, A.: *SafeCap domain language for reasoning about safety and capacity*. In: *Workshop on Dependable Transportation Systems*. IEEE CS, Niigata (2012)
7. Epsilon, <http://www.eclipse.org/epsilon/>
8. The Yices SMT solver, <http://yices.csl.sri.com>
9. The ProB model checker, <http://www.stups.uni-duesseldorf.de/ProB>

Embedded System Platform for Safety-Critical Road Traffic Signal Applications

Thomas Novak and Christoph Stoegerer

SWARCO FUTURIT, Muehlgasse 86, 2380 Perchtoldsdorf, Austria
{novak,stoegerer}futurit@swarco.com

Abstract. Road traffic signals are used to signal information to drivers (e.g., red signal to stop). A controller residing in a local cabinet is managing the traffic signal. Depending on the desired traffic situation, the corresponding traffic signal is switched on or off via the power line. For the time being, there is no programmable logic included in each traffic signal. As a result, there is a single type for each application available e.g., due to different power supply levels or light output. In addition, functionality of traffic signals is limited so that its status information cannot be retrieved.

This paper presents an approach to traffic signals for safety-critical applications based on an embedded system. It includes a presentation of safety-related hardware and a microcontroller with embedded safety-related firmware. The result is a platform to be used in various applications and meeting safety and performance requirements according to standard EN 50556 and VDE 0832.

Keywords: Safety-related embedded system, safety standard, road traffic signals.

1 Introduction

Road traffic signal systems are a common way to control traffic in urban areas via optical traffic signals. Often they are installed at intersections to allow vehicles and pedestrians passing the intersection in a safe way. Such a system typically consists of a road-side controller located in a cabinet at the intersection, traffic signals, and traffic sensors and detector (cf. Fig. 1). Depending on the application, the controller may be linked to a traffic management center.

The system can work in a periodic manner or demand driven. In case of the first option the road side controller uses the same time schedule for each direction throughout a predefined period. For example, each direction gets 45seconds a green signal. The second option means that depending on the traffic flow or someone pressing a button the status of the traffic signal is changed.

In general, a road traffic signal used to signal drivers includes three traffic signal modules (TSM) (red, yellow, green), others only consist of two TSM (e.g., at pedestrians crossings). A TSM is turned on or off when the road-side controller switches power supply on or off. Getting information on the status of a TSM (broken/not broken, on/off) is handled by the level of current. As long as the current is above a predefined value, the TSM is considered to be working.

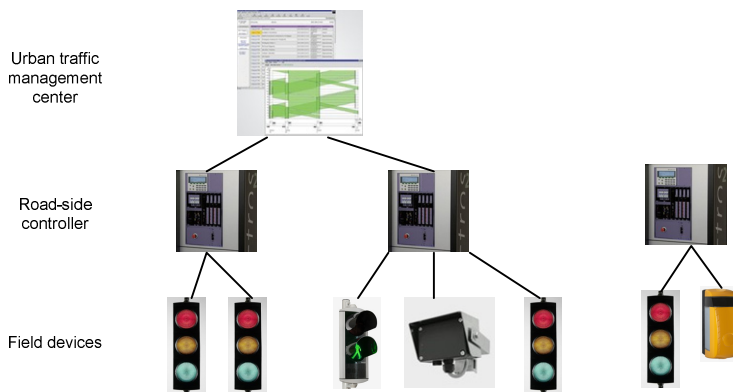


Fig. 1. Road traffic signal system

TSMs across Europe and especially worldwide differ in their size, but also in the electrical characteristics, maximum brightness or way of application. E.g., there are TSMs with 24V or 40V available. Today’s TSM are based on LED technology (cf. Fig. 2). Therefore, the TSM consists of 1-5 LEDs (e.g., depending on the light output or supplier of TSMs) placed on a LED-board. They are driven by a PWM-controller including non-programmable logic. Finally, a red TSM signaling “STOP” is considered to be safety-critical because a malfunction can lead to dramatic casualties. According to European and national standards such as EN50556 [2] or the German standard VDE 0832 [1], road traffic signals have to fulfill a number of requirements relating to the hardware, the software, the application, the integration within an overall system and the engineering process.



Fig. 2. Traffic signal module

The objective of this paper is to present a *LED-based* traffic signal module platform with an *embedded system* that meets the requirements of EN 50556 and of VDE 0832. Benefit of such an approach is that

- a variety of applications with a single platform can be supported,
- an embedded firmware can provide functionality to exchange data with a road-side controller,
- detailed information of failures in traffic signal modules can be sent to a road-side controller,

- a reduction in energy consumption of traffic signal system because the level of current to detect “working TSM” can be reduced significantly,
- the increased functionality (e.g., counting working hours) can support maintenance activities.

The remainder of the paper is structured as follows: Section 2 gives an overview of relevant parts of the standards VDE 0832 and EN 50556, respectively. Section 3 presents a hazard analysis using HAZOP analysis. Section 4, in turn, introduces the embedded system platform architecture derived from the safety and domain specific requirements. In addition, Section 5 proves the architecture by going through a typical use-case. The main steps within the architecture are highlighted. Finally, Section 6 concludes by summarizing the key facts of the work carried out.

2 Standard

The German standard VDE 0832 consists of seven parts (100, 110, 200, 300, 310, 400, 500) and defines requirements on the development, construction, validation and maintenance of road traffic signals. In contrast to generic standards like IEC 61508 [3], this standard is relating to a defined product.

Part 100 of the standard is identical with the German version of EN 50556 including a national foreword. It is relevant for the hardware related part of the embedded platform.

The prestandard part 500 gives requirements on the safety-related firmware of road traffic control systems. This part is mainly referring to IEC 61508 and its requirements. Put succinctly, it specifies non-functional and functional measures for the various lifecycle stages to ensure a certain level of software integrity.

VDE 0832 does not mention risk analysis, but assumes that every critical fault can be either detected within a predefined time frame (e.g., 50ms in case of a TSM) or leads to a safe state not causing any harm to the user.

With all the facts in mind, designing a TSM as an embedded system is a challenging task. First, *safety requirements* coming from the related standard have to be met. These include the detection of safety-critical faults within a predefined time interval. Second, strict *timing requirements* have to be considered (e.g., a blinking TSM means that power is turned on and off every second) to guarantee synchronously running traffic signals. Additionally, the *life-time* of a traffic light system is up to 30 years. Hence, a new TSM must comply with legacy systems to support retrofit.

3 Hazard Analysis

Before starting with the hazard analysis, the scope of hazard analysis has to be specified as addressed e.g., in the IEC 61508-1 life-cycle model. The equipment under control to be looked at is a Traffic Signal Module (TSM). It shall

- be based on an embedded system (microcontroller with firmware),
- provide functionality to communicate with a road-side controller,

- provide support to parameterize e.g., voltage level or maximum level of brightness,
- support additional features such as compensation of degradation of LEDs as added value.

The TSM displays a red signal. The application area of the TSM shall be at an urban intersection. The TSM is controlled by a road side controller via a protocol (e.g., to receive status information from TSM) and the power lines to switch on/off.

Table 1. HAZOP of function “Switch on red traffic signal module”

Guideword	Deviation	(Possible) Cause	Effect
MORE	Brightness level of LEDs higher than expected	LED current too high	Brightness level not as intended
LESS	Brightness level of LEDs lower than expected	LED current too low	Brightness level not as intended
REVERSE	LEDs are switched off, but shall be switched on (“Red signal” is not displayed)	Broken LED	TSM unintentionally switched off
NO	No LEDs are switched on (“Red signal” is not displayed)	No LED current	TSM unintentionally switched off
LATE	LEDs are switched on too late (“Red signal” displayed too late)	Blocking function in the firmware (deadlock)	TSM unintentionally switched off

This hazard analysis is relating to hazards causing harm to the user. Therefore, the display as interface to the user (e.g., a driver on the road) is of interest. Beyond the scope of the analysis is data exchange by means of a protocol between a TSM and a road-side controller because various approaches and implementations are already available [7].

Since the impact of failures to the environment shall be investigated, a hazard and operability (HAZOP) study is a proper approach. A HAZOP study according to Def-Standard 00-58 [4] is a well-defined method to analyze a system at its boundaries. The HAZOP includes pre-defined keywords that are applied to specify the *deviation* from the expected result, the *cause* of the deviation and the (negative) *effect* on the system or environment.

In Table 1 the HAZOP of the function “Switch on red TSM” is presented in a general way. The content of the column “effect” mentions the consequence of a deviation to be seen on the interface to the user. The column “possible cause” includes the typical faults that have to be address by safety measures.

4 Embedded System Architecture

The architecture of the embedded platform has to adhere to product specific and safety requirements. Additionally, constraints posed on the platform coming from the legacy system have to be taken into consideration. In the following, a general hardware and software architecture of the embedded platform is being presented.

4.1 Hardware Architecture

The architecture consists of two major parts as shown in Fig. 3: Power Supply Unit (PSU) and LED unit. The PSU is responsible to supply the LED unit with an adequate level of current and voltage. In addition, it takes care of monitoring the LED unit and providing an interface to a road-side controller. The LED unit, in turn, includes a control loop to supply LEDs with a constant level of current.

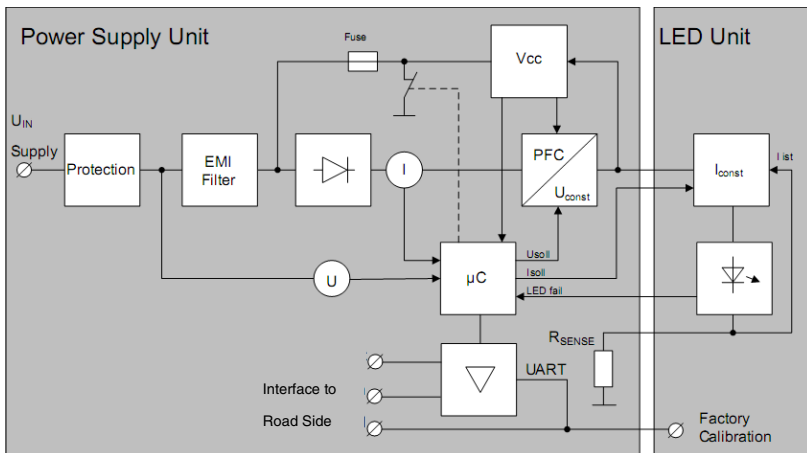


Fig. 3. Hardware architecture of the embedded platform

In detail, the hardware architecture has to support

- the control of the input power and the LED current: After turning on power supply at the road-side controller, the microcontroller starts and is sensing input voltage (U) till it reaches the predefined voltage level. Next, the microcontroller starts the analogue switching regulator and triggers the value for the regulated output voltage (U_{SOIL}). At the same time the LED current is triggered through the low dropout regulator (I_{SOIL}) and measured with the microcontroller functionality (I_{ist}).
- monitoring of LED: The LED voltage (I_{ist}) is monitored by the microcontroller (LED fail).
- entering a safe state by means of a fuse: The microcontroller is supplied with power over a fuse. At startup the fuse will be activated if the microcontroller does not bypass the fuse within a predefined time interval (see dashed line in Fig. 3). In case of any safety-critical failure, the bypass is switched off and deactivates the microcontroller instantly.

4.2 Firmware Architecture

The safety-related firmware is running on a microcontroller embedded in the power supply unit (cf. Fig. 3). It is a 3-tier structure as illustrated in Fig. 4. The lower layer called “driver layer” comprises the hardware related functionality. The application layer includes the management functionality required to run and monitor the system. The application is located at the upper layer and is triggering the required functionality.

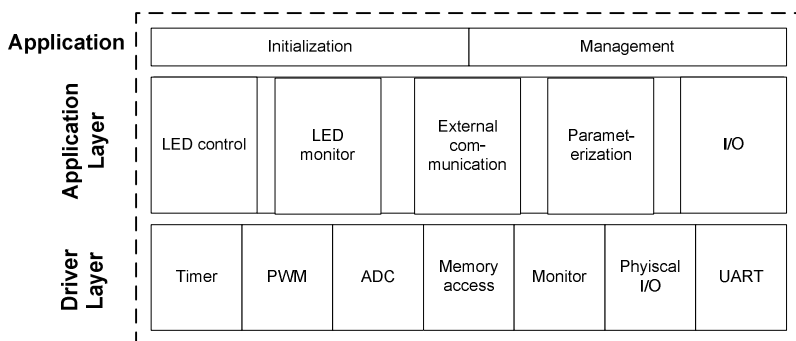


Fig. 4. Firmware architecture

Driver Layer. It consists of seven modules. *Timer* is mainly used to trigger the digitalization of input current/voltage and input current within the *ADC* module. The *PWM* is used to control the LED current. The *Monitor* module includes functionality to trigger the safe state via the *physical I/O* in case of a critical failure. The reason for a safe state or parameter values are written to the non-volatile memory within *memory access* module. Finally, the *UART* is used to send or receive bytes.

Application Layer. It includes five modules. *LED control* and *LED monitor* are called by the application to turn the LED on/off and to check their status, respectively. *External communication* provides the functionality to exchange data with the road-side unit. The parameterization includes all parameters (e.g., voltage level) and functionality to read/write data in the memory. In the end, the *I/O module* gives the opportunity to set/reset the physical *I/O*.

Application. It incorporates a state-machine to trigger the different functionality of the TSM. Moreover, procedures of parameterization and an initialization of component values are included in the layer.

5 Safety Measures

The objective of this section is to show that the safety and safety integrity requirements are met. Consequently, it is demonstrated that adequate safety measures are implemented and hazards mentioned in Table 1 are sufficiently addressed. For that reason, the use-case “switch on red traffic signal module” is taken as an example.

It is assumed that the traffic signal module (TSM) is part of a traffic signal consisting of “red”, “yellow” and “green” and is installed at an intersection. A road

side controller residing in a cabinet next to the intersection is managing the traffic signals. It is connected to each TSM at the intersection via a twisted-pair cable for communication and a power line cable. In case of a safety-critical failure, the safe state is always “switch off” and the level of current is less than 5mA.

Before the deployment the TSM is configured via the factory calibration interface according to the field of application. Typically, the required voltage level and the maximum LED current (i.e., the maximum level of brightness) are set. Moreover, the maximum number of broken LEDs is specified (e.g., more than one broken LED leads to a safe state). The installation and commissioning of the safety-related product is a crucial point [8]. It has to be ensured that the right configuration parameters are uploaded correctly to the microcontroller. A possible solution is outlined in [6].

At operation phase the road-side controller is turning on power to switch on the red TSM. By doing so, the microcontroller is booting. It is checking its memory via error correction codes (hamming code) to ensure the integrity of the memory and to avoid a malfunction of the firmware (e.g., control of LED current not working). Next, the status of the fuse is retrieved via the I/O functionality. If the fuse is operational (i.e., no safe state), the fuse is started to be triggered periodically. In case of a deadlock in the firmware triggering is stopped immediately and the safe state is entered. As a further step, the status of each LED is verified by reading information from the memory. If no LEDs or fewer LEDs than defined are broken, the microcontroller enables the supply of LEDs with the required current.

During the on-phase the microcontroller is measuring the LED current and the voltage. It compares the value with the predefined one set during the commissioning phase. As long as the values are in a specified range, LEDs are working properly. When the LED current drops, a fault of a LED is detected. In case of a non safety-critical failure, information of the new status is sent via protocol to the road-side controller. And LED current is increased to provide same level of luminosity as before. If the number of broken LEDs is above the limit, the safe state is entered. I.e., the microcontroller stops triggering the supply of the LED current and the fuse. Hence, the input current is reduced to a low level and the road-side controller realizes a malfunction in the TSM. In addition, microcontroller writes error code into the memory and sends it to road-side controller in order to ease finding the reason for the failure. The road-side controller turns power off to switch off TSM. Consequently, microcontroller is actively stopping to trigger LED supply and continues triggering fuse. It writes new status into memory.

Aforementioned safety measures and the presented flow of actions are an efficient solution where safety integrity of the TSM (see Table 2) is ensured according to requirements given by VDE 0832.

Table 2. Measures to ensure safety integrity

Integrity of display	Integrity of communication line	Data integrity of microcontroller	Software integrity
Measuring LED current and voltage	Provided by safety-related protocol	Error correcting codes	Triggering fuse periodically

Finally, it must be mentioned that developing safety-related embedded platform for these applications requires additional effort not covered by this work that is also part of a safety development. Whereas the paper presents a general technical solution, an overall safety development needs – among other things – detailed documentation of requirements and design. And a verification and validation approach has to be specified and executed as presented in [5].

6 Conclusion

The paper presented an *embedded system* platform for safety-critical *LED-based* road traffic signal applications in contrast to traditional approaches with a non-programmable logic. A resource efficient and safety-related hard- and firmware design was required tailored to the *safety*, *timing* and *costs* demands. It gives the possibility to support a variety of applications with a *single* platform. Moreover, in new installations a *reduction in energy consumption* of traffic signal system can be reached because input current is not a means of signaling failures any longer. *Increased functionality* (e.g., logging of working hours) can be mentioned as a further advantage of the presented solution.

Integrating intelligence into a TSM might be used in future applications to decentralize a traffic light system or send the status of the TSM (e.g., display remaining time of green phase) directly to cars passing by with the help of car-to-infrastructure communication.

References

1. DIN/VDE: Road Traffic Signal Systems. VDE 0832, part 100-500 (2008-2011)
2. EN: Road traffic signal systems. EN 50556, CENELEC (2011)
3. IEC:Functional safety of electric/electronic/programmable electronic safety-related systems. IEC 61508-1 to -7,edn. 2 (2010)
4. Ministry of Defense: HAZOP Studies on Systems Containing Programmable Electronics, Part 1, Requirements. Standard 00-58, Issue 2 (2000)
5. Tamandl, T., Preininger, P., Novak, T., Palensky, P.: Testing Approach for Online Hardware Self Tests in Embedded Safety Related Systems. In: Proceedings of the 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1270–1277 (2007)
6. Novak, T., Fischer, P., Holz, M., Kieviet, M., Tamandl, T.: Safe Commissioning and Maintenance Process for a Safe System. In: Proceedings of the 7th IEEE International Workshop on Factory Communication Systems (WFCS), pp. 225–232 (2008)
7. Wratil, P., Kieviet, M.: Sicherheitstechnik für Komponenten und Systeme. Hüthig Verlag, Heidelberg (2007)
8. Novak, T., Stoegerer, C.: The right degree of configurability for safety-critical embedded software in variable message signs. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 418–430. Springer, Heidelberg (2010)

Low-Level Attacks on Avionics Embedded Systems

Anthony Dessiatnikoff, Eric Alata, Yves Deswarte, and Vincent Nicomette

CNRS, LAAS, 7 avenue du colonel Roche, 31400 Toulouse, France
Université de Toulouse, INSA, LAAS, 31400 Toulouse, France
{adessiat, ealata, deswarte, nicomett}@laas.fr
<http://www.laas.fr>

Abstract. Improving the security of computing systems embedded into commercial airplanes has become a major concern for the avionics industry. This paper deals with one of the techniques that can be applied to improve the security of such systems: vulnerability assessment. More precisely, this paper presents experiments carried out on an experimental embedded operating system in order to assess vulnerabilities in its low-level implementation layers. The main characteristics of this embedded system, the platform used to carry out our experiments, as well as the first results of these experiments are described.

Keywords: Avionics embedded systems, vulnerability assessment.

1 Introduction

Improving the security of computing systems embedded into commercial airplanes has become a major concern for the avionics industry, malevolence being now considered as a significant potential cause of failures, with potential casualties (airplane passengers). This concern is raised by the expected evolution of on-board computing systems. Indeed, to reduce production and operation costs, these systems tend to:

- Be open to applications and equipment provided by the clients (airline companies) and to open networks (e.g., for digital air traffic control management, airline company information systems, passenger access to the Internet);
- Share computing resources among different applications, possibly with different safety requirements;
- Use COTS hardware and software components.

To satisfy safety requirements, avionics manufacturers have applied formal development and verification methods for decades, at least for the most critical parts of avionics. Currently, these methods target design faults, in particular software bugs. The approach is based on the ARINC 811 decomposition of the information system into several domains with different levels of criticality. They do not address explicitly security concerns, such as deliberately-introduced malicious logic, intrusions, or denial-of-service attacks. This has changed recently,

as malicious actions are becoming a major concern to be taken into account for safety-critical systems.

To improve security of these avionics systems, it is necessary to apply or adapt methods and techniques that have proven their efficiency in other contexts:

- Formal methods for specification, development and verification: such methods are recommended, for instance by the ISO/IEC 15408 Common Criteria (CC), to achieve high levels of assurance¹.
- Security mechanisms and tools: firewalls, VPNs, access control, intrusion detection, etc.
- Vulnerability assessment and countermeasure provision.

This paper focuses on this latest technique. Let us note that vulnerability assessment cannot be covered easily by current formal methods because there is no model suitable for expressing both the desired high-level properties (confidentiality, integrity, availability) and the implementation of subtle hardware mechanisms (control of address space [1], interruptions, handling, or hardware management functions such as ACPI [2], or even protection mechanisms [3]). For instance, formal approaches that apply successive refinements of models (e.g., B method [4]) cannot descend to the level of COTS processors [5]. In addition, formal verification is based on assumptions about the underlying layer that may be wrong in operating environment and can lead to vulnerabilities. In fact, experience shows that even formally proven systems may have exploitable vulnerabilities ([5], [6]). It is therefore necessary to analyze vulnerabilities, even at the lowest abstraction levels, and provide solutions able to counter them.

This paper presents some experiments run in the context of the French ANR project SOBAS (*Securing On-Board Aerospace Systems*), using an experimental embedded avionics operating system, compliant with the current avionics specification standards. This operating system, described in Section 2, was supplied by Airbus France, partner of the project. Section 3 gives an overview of the hardware platform that we use to inject low-level attacks to this embedded system. Section 4 describes the experiments we have run up to now as well as the corresponding results. Section 5 concludes the paper.

2 Overview of the Avionics Embedded Operating System under Study

The embedded operating system we use for our experiments has been designed and developed according to the ARINC 653 avionics standard (*Avionics Application Software Standard Interface*). The ARINC 653 is a standard interface for time and space partitioning of embedded systems resources. It is compliant with the IMA (*Integrated Modular Avionics*) trend. The main objective of IMA is to simplify the integration of avionics software, by proposing the use of

¹ These criteria enable to evaluate the security of a product according to predefined profiles.

a set of shared hardware and software resources. According to the IMA trend, each avionics function is not implemented any more on a dedicated computer but may share hardware or software computing resources with other functions. These computing resources communicate through a redundant deterministic Ethernet network called AFDX.

Furthermore, each computing resource (implemented according to the ARINC 653 API) must provide an efficient temporal and spatial partitioning so that it can host applications with different levels of criticality.

The embedded system we studied is thus composed of :

- The kernel, which is the core software; it schedules the different partitions and enforces their memory and spatial isolation.
- The application partitions, which are tasks with a low level of criticality.
- The system partitions, which are tasks with a high level of criticality.
- The system specific functions such as drivers, downloading code, debug and tests functions.

The kernel is designed to schedule partitions under real-time constraints through the use of three important hardware components: MPIC (*Multicore Programmable Interrupt Controller*), caches and MMU (*Memory Management Unit*). The MPIC and the MMU are used by the kernel in order to implement the spatial and temporal isolation between the different partitions, whereas the caches are used to improve the memory access speed. Furthermore, the allocation of memory and time for the partitions is made statically at build-time, through a specific configuration tool.

More precisely, the kernel starts by initializing itself and then the different partitions, which are to be executed in a predefined static order. The MPIC generates an interruption each time a dedicated counter reaches a specific value. The kernel uses the MPIC to implement the temporal partitioning between applications: an interrupt is generated each time a time slot allocated for a particular partition ends up. The dedicated counter is then set to zero, the context of the current partition is backed-up and the context of the next partition to be executed is restored. Regarding the memory isolation, the MMU is used for spatial partitioning: it controls the memory accesses in such a way that a partition cannot read or write application data used in the context of another partition.

3 Platform Description

We realized our experiments on a P4080 hardware platform provided by Freescale. This platform has been purposely chosen because it is included in a list of hardware platforms currently under study in the SOBAS project. More precisely, in this project, an experimental platform has been designed around the P4080 using, in addition, a CodeWarrior IDE, a JTAG probe and a laptop.

The P4080 is a platform that can be used in different ways. It is a powerful platform and it has typically been designed to be integrated in powerful networking equipments, since it includes important features to accelerate packets processing and filtering.

We used CodeWarrior as an Integrated Development Environment (IDE) that provides a visual and automated framework to accelerate the development, debugging, compilation, uploading and execution of complex programs on different hardware platforms. This tool is very useful in our experiments to observe the behavior of the software as well as to inject low-level attacks on our P4080 platform. CodeWarrior is executed on a laptop connected to the P4080 through a JTAG interface.

There are two different kinds of JTAG probe: USB and Gigabit. The USB probe is slower than the Gigabit probe, which uses Ethernet port. For our experiments, we have used the USB probe so far.

4 Experiments

For the rest of this paper, we are considering the following four attack assumptions: (1) an attacker has no physical access at run-time to the systems; (2) some actors are trusted: pilots, maintenance, crew; (3) many other actors can be potentially malicious: passengers, personnel of airline companies, equipment manufacturers; (4) consequently, we consider the possibility that a non-critical application partitions is malevolent and may carry out malicious actions in order to corrupt critical partitions or even the kernel itself.

We are running experiments in order to assess vulnerabilities in such a compliant ARINC 653 system. Our objective is to elaborate attacks that target 1) the core functions of the system and 2) the fault tolerance mechanisms implemented for safety reasons in the system. The core functions are the standard components of an avionics operating system, such as processor, memory management, process management, scheduling, communications, time management, cryptography and ancillary functions. The fault tolerance mechanisms are usually classified into fault handling and error handling functions [7]. They are an interesting target for an attacker because their corruption may be an efficient way, for instance, to provoke a denial of service of the system. As an illustration, if an attacker is able to corrupt a critical functions (replicated on several modules for safety reasons) in such a way that the all redundant copies always disagree, it can provoke such a denial of service.

In the next subsections, we present three different attacks that we carried on our experimental kernel. Two of the attacks target the core functions of the system, more precisely, the memory management and the time management. The third attack target the fault tolerance mechanisms.

4.1 Attacks on Memory Management

A corruption of the memory management may allow a malevolent partition to access sensitive data used by the kernel or by other partitions. We describe in the following such an example. The kernel configures the P4080 platform using a memory area of 16 MBytes, called the *Configuration Control and Status Registers* (CCSR).

This region contains sensitive registers used to configure various platform components. After analyzing the source code of our experimental kernel, we have discovered that the read/write accesses to the CCSR area region were authorized for the kernel but also for the partitions. This configuration was chosen deliberately so that the different partitions can directly use network peripherals (and network peripherals configuration) without any intervention of a shared driver. However, this design is particularly risky as it allows any malicious partition to fill registers of the CCSR area with random values. For our experiments, we injected some purposely chosen values in this area, from the malevolent partition, in order to provoke a denial of service of the platform. We could provoke such a denial of service using the Security Engine (SEC) and the Run Control/Power Management (RPCM). In the following, we only describe the experimentation using SEC.

The security engine is a cryptographic hardware accelerator, implementing RSA, DES, AES, SHA algorithms and other cryptographic functions. Its configuration is made through a dedicated memory region in the CCSR. The security engine is managed by the *security monitor* component of the P4080. The security monitor is in charge of checking the system for errors and controlling the cryptographic keys used to securely boot the system. If an error is raised when checking the system at startup-time, the security monitor enters into a failed state, which blocks the cryptographic keys and sets them all to zero. At this stage, the system may restart if the security monitor was configured to do so after such an error. This configuration is made by setting some registers of the CCSR area. As any partition can access in write mode the CCSR area, a malicious partition is perfectly able to purposely modify these registers and then provoke an error in order to restart the P4080 platform. These experiments are interesting because they actually show that the CCSR area is a critical region of memory that must be protected. This region must only be writable by the kernel (with supervisor privilege). Otherwise, a malicious partition is perfectly able to provoke a denial of service of the system.

These protections on the CCSR area can be configured through the MMU, which actually offers the possibility to precisely set access rights for different regions of memory.

4.2 Attacks on Time Management

An attack on time management consists, for a malevolent partition, in being able to modify the execution duration of a critical partition. These execution durations are very important in such hard real-time embedded systems and any modification of these durations is, at least for safety, not acceptable.

The execution duration for each task is based on the WCET (*Worst Case Execution Time*) computation. At the end of each execution duration, an interrupt is generated so that a new partition can be scheduled. To modify the execution duration of a critical partition from a non-critical malevolent partition, we identified three possibilities: (1) modify the CCSR area, in which the execution durations for all the tasks are specified; (2) generate an interrupt before the

end of the execution duration of the critical task; or (3) execute kernel code to delay as much as possible the end of the execution the non-critical task (and as a consequence, to reduce the execution duration of the next critical task, if we suppose that the critical tasks are scheduled immediately after the non-critical task).

We do not detail the first possibility because it consists in an attack targeting the memory management, which has already been explained in the previous subsection. The second possibility consists in being able to generate an interrupt before the end of the execution duration of a particular task. The only possibility to generate such an interruption is to modify the MPIC, which requires supervisor privilege. It is thus impossible to carry out this attack on our platform. The third possibility consists, for a non critical task, in provoking the execution of kernel code just before the end of its execution duration. Since the kernel code is executed with supervisor privilege and as consequence, cannot be interrupted, it is possible for a task T1 to increase its own execution duration, and as a consequence, to reduce the execution duration of the following task T2 to be scheduled. If T1 is non-critical and T2 is critical, this attack allows a non-critical task to modify the execution duration of a critical task. The main problem of this attack is to identify and call some kernel code that is actually able to significantly exceed the execution duration of a partition. Another problem is that the only way for a partition to execute kernel code is to provoke an exception. Thus, we carried out experiments in which we purposely provoke exceptions at the end of a non-critical partition. More precisely, we implemented a loop specifically dedicated to stop just before the end of the duration partition, and then we provoked an exception by using an illegal instruction.

For our experiments, we used two partitions P0 and P1. P0 lasts 600 microseconds and P1 lasts 500 microseconds. We modified the P0 partition to provoke an exception at the end of its execution duration in such a way that a kernel function is called. In this experiment, the consequence was the direct reduction of execution duration of the partition P1 to 460 microseconds. This may have serious consequences or not, depending of the applications. Anyway, this experiment showed that there is actually a possibility for a non-critical partition to modify the execution duration of a critical partition. Different countermeasures can be imagined in order to prevent a partition from delaying its own execution duration, either by modifying the kernel design or by introducing margin in the calculation of the execution durations of the user partitions. Regarding the kernel modification, it may consist in allowing the kernel to receive some interrupts (especially the timer interrupts) or strongly the duration of exception management in the kernel.

4.3 Attacks on Fault Tolerance Mechanisms

The fault tolerance mechanisms can be classified in two categories: error handling (error detection and then rollback, rollforward or compensation recovery) and fault handling (diagnosis, isolation, reconfiguration, reinitialization) [7]. A non critical malevolent partition may target the fault tolerance mechanisms in

order to make these mechanisms constantly detect errors and restart the system, provoking in this way a denial of service.

A typical fault tolerance mechanism is the exception management of the kernel. This mechanism can provoke a module restart when errors considered as unrecoverable occur. If a malevolent partition is able to generate this kind of exceptions, it may provoke the module restart at any moment. In order to test the exception management, we decided to implement a *crashme*² program that executes random instructions. Such a program could be used by a malicious partition to test if some sequences of instructions are able to force a core module to stop or restart, or provoke a kernel crash for instance. We created a *crashme* program in a partition which generates random instructions and executes them. The *crashme* was executed during 1 043 806 application cycles (each application cycle is 5ms). A hundred random instructions were executed during each cycle. This experimentation showed that, among the set of 22 exceptions, only one of them represents more than 99% of the exceptions raised, seven of them represent only 1% and the other 14 exceptions are never called.

Moreover, we noticed that 16 out of 22 exceptions provoked a kernel restart, kernel stop or a module restart. It means that a non-critical partition executing such *crashme* program has many chances to stop or restart the module, provoking in this way a denial of service since the module stops delivering services for the aircraft. This experiment was particularly interesting because it put the emphasis on the fact that the current exception management in the experimental operating system under study must be improved in such a way that it drastically reduces the number of situations that lead to a module stop or restart.

For that purpose, the origin of the exception has to be precisely identified so that not to restart the whole module or the kernel unless it is strictly necessary.

5 Perspectives

In this paper, we have described a set of experiments carried out on an experimental avionics embedded system, in order to assess some low-level vulnerabilities : 1) the modification by non-critical partition of some sensitive areas of memory ; 2) the modification of the duration of a partition under some circumstances and 3) the abuse of the exception management in order to restart the kernel. These first experiments are promising as they permit to identify a set of vulnerabilities in the kernel and a set of improvements that should be implemented in order to increase the security level of the global system.

We are currently investigating other experiments. The first one consists in abusing the update of data in the flash of the P4080. Thus process is performed thanks to the cooperation between two user applications. We plan to check whether the corruption of one of these applications could provoke the flashing of purposely corrupted data. A second attack consists in using a core to execute malicious code. Actually, only one core is currently used in our experimental platform but we want to test the level of difficulty that is required to activate

² <http://crashme.codeplex.com>

another core. If such an activation can be easily done by a malevolent partition, then some malicious code could be injected and executed. The third attack consists in considering two P4080 platforms, each one executing a set of partitions and communicating through an AFDX network. We plan to test whether it is possible or not for a malevolent partition of one of the platform, to send malicious data through AFDX communication to a partition of the other platform, in order to provoke its corruption.

Acknowledgements. This study is partially supported by DGA (the French Armaments Procurement Agency) with scientific supervision by Véronique Serfaty, and by the French ANR Project SOBAS³ (Securing On-Board Aerospace Systems).

References

1. Lacombe, E., Nicomette, V., Deswarte, Y.: Enforcing Kernel constraints by hardware-assisted virtualization. *Journal of Computer Virology* 7(1), 1–21 (2011)
2. Duflot, L., Levillain, O.: ACPI et routine de traitement de la SMI: des limites l’informatique de confiance. In: *Symposium sur la Sécurité des Technologies de l’Information et des Communications (SSTIC)* (June 2009)
3. Lone Sang, F., Lacombe, E., Nicomette, V., Deswarte, Y.: Exploiting an I/OMMU vulnerability. In: *International Conference on Malicious and Unwanted Software (MALWARE 2010)*, Nancy, France, October 19-20, pp. 9–16 (2010)
4. Abrial, J.-R.: Extending B Without Changing it (for Developing Distributed Systems). In: *1st Conference on the B method, Putting into Practice Methods and Tools for Information System Design*, pp. 169–190. Institut de Recherche en Informatique de Nantes, France (1996)
5. Jaeger, E., Hardin, T.: A few remarks about formal development of secure systems. In: *11th IEEE High Assurance Systems Engineering Symposium (HASE)*, pp. 165–174 (2008)
6. Sibert, O., Porras, P.A., Lindell, R.: The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems. In: *IEEE Symposium on Security and Privacy* (1995)
7. Avizienis, A., et al.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1(1), 11–33 (2004)

³ <http://projects.laas.fr/sobas/>

Safety Cases and Their Role in ISO 26262 Functional Safety Assessment

John Birch¹, Roger Rivett², Ibrahim Habli³, Ben Bradshaw⁴, John Botham⁵,
Dave Higham⁶, Peter Jesty⁷, Helen Monkhouse⁸, and Robert Palin⁹

¹ AVL Powertrain UK Ltd, Basildon, UK

² Jaguar Land Rover, Coventry, UK

³ University of York, York, UK

⁴ TRW Conekt, Solihull, UK

⁵ Ricardo UK Ltd, Cambridge, UK

⁶ Delphi Diesel Systems

⁷ Peter Jesty Consulting Ltd, Tadcaster, UK

⁸ Protean Electric Ltd, Surrey, UK

⁹ MIRA Ltd, Nuneaton, UK

Abstract. Compliance with the automotive standard ISO 26262 requires the development of a safety case for electrical and/or electronic (E/E) systems whose malfunction has the potential to lead to an unreasonable level of risk. In order to justify freedom from unreasonable risk, a safety argument should be developed in which the safety requirements are shown to be complete and satisfied by the evidence generated from the ISO 26262 work products. However, the standard does not provide practical guidelines for how it should be developed and reviewed. More importantly, the standard does not describe how the safety argument should be evaluated in the functional safety assessment process. In this paper, we categorise and analyse the main argument structures required of a safety case and specify the relationships that exist between these structures. Particular emphasis is placed on the importance of the product-based safety rationale within the argument and the role this rationale should play in assessing functional safety. The approach is evaluated in an industrial case study. The paper concludes with a discussion of the potential benefits and challenges of structured safety arguments for evaluating the rationale, assumptions and evidence put forward when claiming compliance with ISO 26262.

Keywords: Safety cases, safety arguments, ISO 26262, automotive safety.

1 Introduction

Critical functions in road vehicles are increasingly being implemented using electrical and/or electronic (E/E) systems. The malfunctioning behaviour of these systems can contribute to the safety risk to the vehicle occupants and/or other road users. As such, it is necessary to provide assurance that any unreasonable residual risks have been

avoided. The safety standard ISO 26262 has been developed to address this necessity by providing guidance, in the form of requirements and processes, for avoiding unreasonable residual risk caused by the malfunctioning behaviour of E/E systems [1]. Like many safety standards that cover complex software-based systems, ISO 26262 defines requirements for the creation of work products i.e. outputs from the safety lifecycle, and leaves it to the developers to interpret these requirements in the context of their products [2]. In order to provide a product-specific justification, compliance with the ISO 26262 standard requires the development and evaluation of a safety case for the safety-related items. The standard defines an item as a “*system or array of systems to implement a function at the vehicle level*” [1]. In order to justify freedom from unreasonable risk, a safety case argument should be developed in which the safety requirements are shown to be complete and satisfied by the evidence generated from the ISO 26262 work products. However, the standard does not provide practical guidance on the development and review of the safety argument, nor does it describe how the safety argument should be evaluated in the functional safety assessment process.

In this paper, we build on the experience of the authors in developing and evaluating safety cases in the context of ISO 26262. We examine the significance and nature of the product-based safety rationale within the argument and the role this rationale should play in assessing functional safety. The paper also builds on existing work on safety cases across different domains [3-5], and in the automotive industry in particular [6], [7], taking into account issues related to product-based and process-based assurance [8], the process of compliance [9] and assessment of confidence [10], [11].

The paper is organised as follows. In Section 2, we categorise and analyse the main argument structures of a safety case and the relationships that exist between the safety case and the ISO 26262 functional safety assessment. The approach is evaluated in an industrial case study in Section 3. In Section 4, we discuss the potential benefits and challenges of structured safety arguments for evaluating the rationale, assumptions and evidence put forward when claiming compliance with the ISO 26262 standard.

2 Safety Argument Categories in ISO 26262

ISO 26262 defines a safety case as an “*argument that the safety requirements for an item are complete and satisfied by evidence compiled from work products of the safety activities during development*” [1]. That is, the argument should play a central role in justifying why the available evidence, in the form of work products (e.g. design and analysis artefacts), has achieved a set of safety requirements and, therefore, why an acceptable level of safety has been achieved. Compliance with ISO 26262, based on the normative parts of the standard, mandates the satisfaction of a specific set of objectives by the generation of a concrete set of work products. As a result, all E/E systems that are compliant with the standard share a common safety argument structure linking the top-level safety requirements to the available evidence. Unfortunately, this common argument structure is implicit and is not documented in the standard.

2.1 Implicit Safety Argument in ISO 26262

The implicit safety argument in ISO 26262 is centred on the following chain of reasoning (Fig. 1). A sufficient and an acceptable level of safety of an E/E system is achieved by demonstrating absence of unreasonable risk associated with each hazardous event caused by the malfunctioning behaviour of the item (other hazard causes are outside the scope of the standard). This is achieved by defining safety goals to avoid unreasonable risk through the prevention or mitigation of the identified hazardous events. A hazardous event is the occurrence of a hazard in particular operational situations. Each hazardous event is assigned an Automotive Safety Integrity Level (ASIL), based on the combination of three parameters: severity (extent of human harm), probability of exposure (to operational situations) and controllability (ability for persons at risk to take action to avoid harm). Claims are then asserted that each safety goal is satisfied by the development of a functional safety concept. The functional safety concept specifies safety measures within the context of the vehicle architecture, including fault detection and failure mitigation mechanisms, to satisfy the safety goals. Two further hierarchies of claim are defined for asserting how the functional safety concept is adequately refined and satisfied by a technical safety concept and hardware and software components (again to the required ASIL). As a result, the implicit argument follows a hierarchy of claims that can be grouped as follows:

- Safety Goals (hierarchy 1) – the vehicle in its environment;
- Functional Safety Requirements (hierarchy 2) – the vehicle and its systems;
- Technical Safety Requirements (hierarchy 3) – the E/E system; and
- Hardware and software requirements (hierarchy 4) – component and part level.

For each hierarchy, ISO 26262 prescribes evidence, in the form of work products, for substantiating these claims. Additionally, the standard identifies methods for generating these work products in accordance with the required ASIL. For example, in order to substantiate a claim that the technical safety requirements have been correctly implemented at the hardware-software level, evidence should be provided through methods such as a requirements-based test, fault injection test or back-to-back test (Table 1, Part 4). This evidence should be captured in an Integration Testing Report (Work Product 8.5.3, Part 4).

The implicit safety argument in ISO 26262 has two main categories of claim: product claims and process claims. Based on the hazard analysis and risk assessment, the product claims focus primarily on the safety goals and safety requirements (i.e. specifying and demonstrating behaviour which is free from unreasonable risk). The process claims focus on the adequacy of the organisations, people, lifecycles, methods and tools involved in the generation of the work products. The nature of these process claims and the rigour of the evidence needed to support them vary with the ASIL assigned to the safety goals and their corresponding safety requirements (i.e. high levels of risk require high levels of process rigour).

Compliance with ISO 26262 and the evaluation of the above implicit argument is demonstrated, in part, using two types of confirmation measures: functional safety audit and functional safety assessment. The requirements for both, and the necessary

independence, are specified in Part 2 of the standard. The functional safety *audit* is concerned with reviewing the implementation of the *processes* required for functional safety. Functional safety *assessment* is concerned with making a judgement on the functional safety achieved by the item and hence is concerned with the characteristics of the *product*. The assessment includes evaluating the work products specified in the item’s safety plan, the required processes (i.e. the functional safety audit) and the appropriateness and effectiveness of the safety measures that are implemented.



Fig. 1. Implicit ISO 26262 Safety Argument Structure

2.2 Product-Specific Safety Rationale

A legitimate question at this point should be: why is it necessary to document the above safety argument if it is common to all items compliant with ISO 26262? What is the added value of developing, reviewing and maintaining this common safety argument? In this paper, we contend that the challenge does not lie in merely capturing this common, implicit, argument. Instead, most of the effort should focus on justifying, through an explicit argument structure, how one hierarchy of claims, e.g. concerning absence of unreasonable risk, is supported by another hierarchy of claims, e.g. safety goals that address any unreasonable risk (Fig. 1). These sub-arguments should capture the product-specific safety rationale which typically varies from one item to another. That is, although the overall structure of the argument is stable (i.e. assessment of hazardous events and specification, development and assessment of safety goals and safety requirements), the assurance challenge lies in providing product-specific rationale, assumptions and justifications for why, given an operational environment, a vehicle configuration and the condition of other vehicle systems, the available evidence is sufficient to support the asserted claims. Typically, these claims, and their corresponding arguments and evidence, are the focus of the functional safety assessment process as they address the product-specific safety rationale that is often associated with unique characteristics of the system and its environment.

A claim is typically made that the absence of unreasonable risk of a hazardous event has been addressed by conforming to a safety goal. However, it would be naïve to define a safety goal as simply a negation of a hazardous event and simply to assign an ASIL to that safety goal. Although this approach is arguably valid from the perspective of literal ISO 26262 compliance, it is simplistic as it limits risk mitigation to

reducing the probability of the hazardous malfunction. Other risk reduction strategies related to reducing severity, improving controllability and/or reducing exposure (typically through a measure “external” to the item, which can be another E/E system) can be taken into account. For example, if a safety goal stipulates that the system shall transition to a safe state in the presence of faults that could otherwise cause the corresponding hazardous event, then an argument and evidence for why the specified safe state is considered to be adequately safe should be provided. This can be achieved by justifying that, were the vehicle behaviour in the safe state to be subject to ISO 26262 hazard classification criteria, then it would be classified ‘QM’ (Quality Management). QM in ISO 26262 denotes a risk that does not require the satisfaction of any specific safety requirements, thereby implying that the level of risk is reasonable and no further risk reduction is necessary. The main claim here would be that the *residual risk* associated with the hazardous event, after achieving the safety goal, has been reduced to a level that is *reasonable*. The subsequent argument used to support such a claim would then need to explicitly assert which risk parameters (‘controllability’, ‘severity’ or ‘exposure’) would be reduced if the residual risk were classified in this way.

A typical approach may be to provide an argument that some reconfiguration or degradation scheme is capable of placing a system into a safe state such that the controllability of any reaction, e.g. to an undemanded drive torque, is effectively C0 (controllable in general) whereas the hazardous event itself will have been classified with the controllability parameter taking a value of C1, C2 or C3. Another approach may be to place a system in a safe state by preventing a vehicle exceeding a speed threshold upon detection of a fault that can cause the hazardous event such that the exposure parameter that could be associated with the safe state is effectively E0 (incredible). Such reasoning is product-specific and the implicit safety argument in ISO 26262 does not prescribe any product-specific safety rationale.

The safety argument structure in Fig. 1 includes references to five product-specific safety rationale sub-arguments. These sub-arguments should provide justification for the inferential transition from one hierarchy of safety claims to another. For instance, the functional safety concept rationale argument should include a justification for why the deployment of safety measures such as fault detection, failure mitigation and/or driver warnings should lead to the satisfaction of the corresponding safety goals.

3 Industrial Case Study

This case study is based on a typical electric vehicle architecture (technology-specific details have been abstracted for reasons of commercial sensitivity), in which a basic Item Definition and hazardous event are considered. The purpose of the case study is to examine the product-based safety rationale arguments, discussed in Section 2, for the corresponding Safety Goal and Functional Safety Concept.

3.1 Item Definition

The Item Definition is shown in Fig. 2. The pertinent nominal operation is as follows:

- Driver requests positive longitudinal vehicle acceleration by depressing accelerator pedal
- Accelerator pedal provides a low voltage electrical signal to indicate pedal position to the Controller
- Controller reads this pedal signal and places a corresponding torque demand on the High Voltage Power Inverter (HVPI) via the Controller Area Network (CAN)
- HVPI converts a certain quantity of electrical energy from the High Voltage Battery to high voltage electrical power supplied to the Electric Machine, according to the torque demand from the Controller
- High voltage electrical power supplied to the Electric Machine induces a mechanical torque in the Electric Machine, which is transferred through the transmission to the vehicle’s rear wheels.

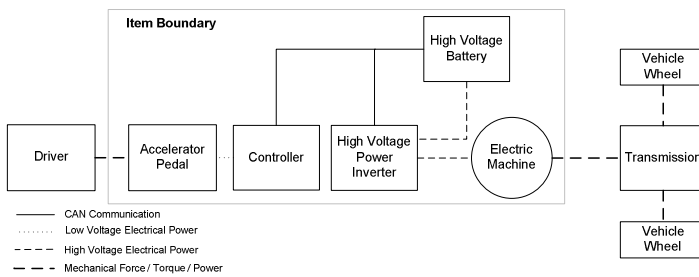


Fig. 2. Electric Vehicle Propulsion System

3.2 Hazard Analysis and Risk Assessment

This case study focuses on the Hazardous Event ‘*Unintended vehicle acceleration during a low speed manoeuvre amongst pedestrians*’, which is classified as ASIL B based on values of E3 (medium probability), S2 (severe and life-threatening injuries, survival probable), C3 (difficult to control or uncontrollable) for the Exposure, Severity and Controllability parameters respectively. The rationale for this classification requires a detailed description of the vehicle, the operational and environmental constraints and peer systems and as such it has not been included for brevity.

3.3 Safety Goal and Its Rationale Argument

The safety goal that has been defined to address the risk associated with the Hazardous Event is ‘*Vehicle positive longitudinal acceleration shall not exceed driver demand by $> 1.5 \text{ m s}^2$ for longer than 1 s*’. However, the question that a safety assessor may rightly ask is why by meeting this safety goal is unreasonable risk avoided? It is not typical within industry for the answer to questions of this type to be documented, but doing so should help the engineer to be clear about why the safety goal achieves freedom from unreasonable risk, and to communicate that to the safety assessor.

The argument for this particular case study, presented in Goal Structuring Notation (GSN) [12] in Fig. 3, is based on improving *controllability*; specifically if the

unintended acceleration is kept below the stated threshold, the driver is able to slow and stop the vehicle before a collision with the pedestrian occurs. Within this argument, the ‘Absence of Unreasonable Residual Risk’ strategy is generic, and could be applied to any safety goal, whereas the ‘Residual Risk Controllability Classification’ strategy is specific to this particular safety goal.

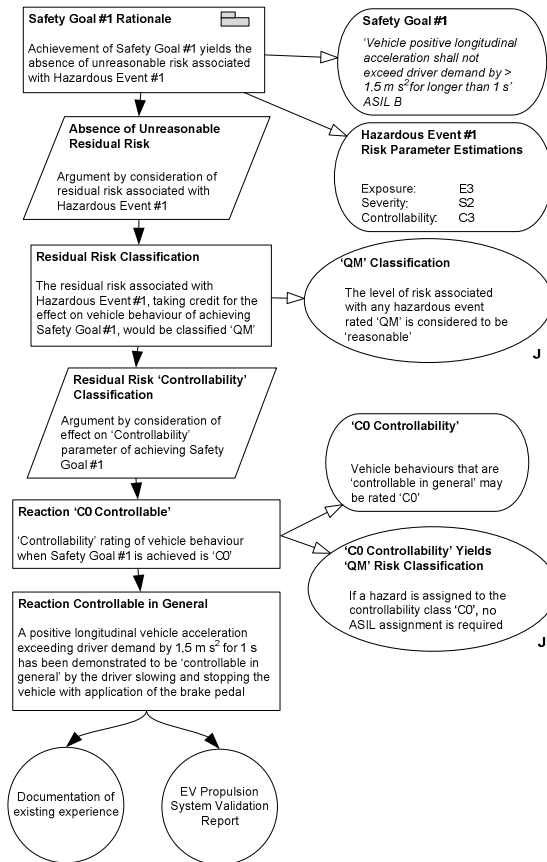


Fig. 3. Safety Goal Rationale Argument

3.4 Functional Safety Concept and Its Rationale Argument

The functional safety concept that has been chosen to achieve the safety goal, named ‘Distributed detection and mitigation of torque errors’, is based on degradation; whereby all faults that can lead to excessive acceleration are detected within an acceptable time interval. On detection of a fault, the vehicle acceleration is limited to a value below that specified in the safety goal. The concept is based on the assertion that only malfunctioning behaviour of the Item that can violate the safety goal (which is specified in terms of *vehicle-level* behaviour; *acceleration*) is the delivery of

excessive *torque* to the Transmission; behaviour which is specified at the *Item-level*. The concept features are as follows (Fig. 4):

1. Detection of all faults that would otherwise lead to excessive torque delivery:
 - (a) Controller detects accelerator pedal faults by comparing and arbitrating between the outputs from two independent pedal position measurement sensors
 - (b) Controller self-detects torque-request errors by comparing its final torque request to the HVPI (output) with the accelerator pedal position (input)
 - (c) HVPI self-detects torque-demand errors by comparing the quantity of high voltage electrical power supplied to the Electric Machine (output) with the torque request from the Controller (input)
2. Upon detection of errors, outputs are electronically ‘limited’ to a fixed value that ensures that the magnitude of excessive torque delivered to the Transmission is below that required to violate the safety goal’s acceleration criteria.

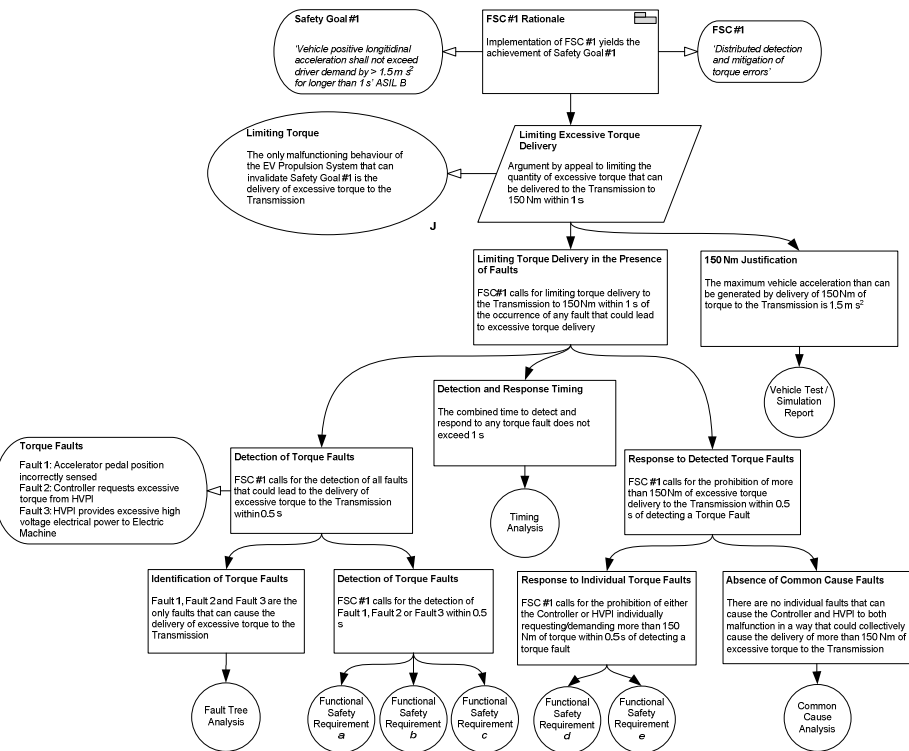


Fig. 4. Functional Safety Concept Rationale Argument

Typically, a company would document the failure modes of the concept in an analysis report, e.g. using Failure Mode and Effects Analysis (FMEA), and manage safety goal and functional safety requirements in a requirements database. It would also have vehicle test reports or simulations demonstrating that the safety goals had been met. However, the rationale explaining how this evidence fits together is not often

documented. This means that whoever performs the Functional Safety Assessment has to deduce this for themselves by ‘reading between the lines’, and for complex and highly interconnected systems, tenuous leaps may need to be made. The added value of formally documenting the rationale, as in Fig. 4, is not only that it helps the engineers to identify potential deficiencies in the safety argument, but also that it eases the subsequent task of performing the Functional Safety Assessment, and may highlight the need for further work.

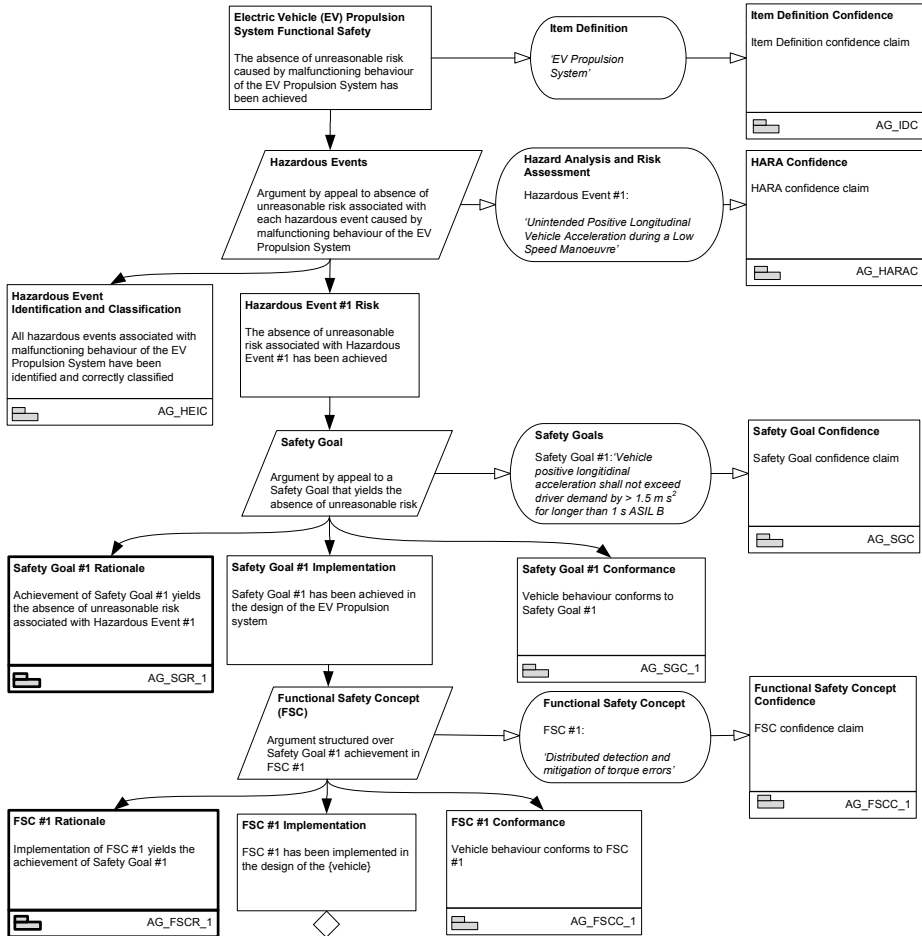


Fig. 5. Item Functional Safety Argument

3.5 Item Functional Safety Argument

The two arguments presented in Fig. 3 and Fig. 4 can be referenced as ‘Away Goals’ [12] within the complete safety argument for the Item (Fig. 5). Other structures within the complete safety argument should include confidence-based process claims that

refer to the ASIL-specific processes used to develop the work products. The complete argument would also require further development to justify how the functional safety concept has been implemented by the chosen technical safety concept and subsequently by the hardware and software safety requirements. Although this has not been the focus of this paper, it has been found that the argument structure at the level of safety goals and functional safety concept can be successfully repeated at lower levels, with the capability to partition the argument to represent the distributed development commonly seen between a vehicle manufacturer and its suppliers.

4 Analysis and Discussion

Without a clear safety argument structure, a checklist approach to safety assurance, based on the creation of work products and requirements traceability, tends to be used. Important as this is, the rationale behind the requirements is often not documented. An important aspect of capturing the product-based safety rationale is that it helps the engineers identify potential deficiencies in the argument in a timely manner and supports the subsequent task of performing the Functional Safety Assessment. In this section, we reflect on the insights gained from different engineering perspectives.

4.1 Original Equipment Manufacturer (OEM) Perspective

Typically, a large list of safety requirements and work products is presented to an OEM, i.e. the vehicle manufacturer, for which there may be traceability to the safety goals but no, or only tenuous, basis for understanding whether and how the safety goals have been fully satisfied. Consequently the adequacy of the deliverables can only be determined by extensive question and answer sessions. This often reveals that the important safety rationale is not documented and only exists in the heads of the engineers. It also often reveals that there are many undocumented assumptions which need to be validated and would be better treated as safety requirements. Where the approach presented in this paper is adopted, engineers gain a deeper understanding of the system they are designing. Further, documentation is generated at a more appropriate lifecycle stage to enable effective and timely assessment.

Because of the hierarchical nature of the explicit safety argument, and the observation that its structure repeats between levels, the argument lends itself to being ‘split’ between organisations. For example, an OEM may typically develop the safety argument down to, and including, the level of functional safety concept. The supplier responsible for developing the technical safety concept and hardware and software safety requirements can then develop the relevant downstream rationale in a similar manner to the OEM in order to justify the safety requirements they have developed.

4.2 Supplier Perspective

E/E system suppliers are heavily dependent on requirements received from the OEM, as the OEM has a complete view of the vehicle, its systems and their dependencies.

However, by developing an E/E system and hardware and/or software components, a supplier will generally own a high proportion of the faults that can contribute to hazardous events. As such, suppliers will be responsible for the design requirements, safety analysis and verification of the E/E system to support the claim that vehicle level safety will be assured and that the requirements of the functional safety concept have been achieved. With this partitioning of responsibilities comes the need to demonstrate accountability i.e. the need for suppliers to provide a safety argument to justify that their design/implementation supports certain safety goals at the vehicle level. A structured argument provides this much needed visibility between parties at the different assessment stages.

Further, suppliers traditionally develop common E/E platforms prior to OEM engagement. For example, a supplier developing an engine management system for future vehicle emission legislation will identify requirements years before involving OEMs. It is important that any safety-related component/element, which is developed without a specific application context, is assessed. The supplier will need to capture assumptions, most likely based on previous experience, and possibly in isolation. These assumptions and the safety rationale are very well suited to an argument structure that clearly identifies product safety claims in relation to assumed hazards, safety goals and concepts. A clearly defined argument structure improves the engagement of customers with new applications not only to provide the safety justification but also to identify assumptions that require confirmation, redress and also the allocation of risk mitigation responsibilities to customers when needed.

4.3 Safety Assessor Perspective

In the infancy of ISO 26262, early project assessments have been based solely on work products and processes. This has resulted in lengthy protracted assessments, trawling through documentation and relying heavily on interviews to discover undocumented rationale. This has highlighted the need to have a safety case with a clear structure and purpose. It has also been found that it is both possible and beneficial to assess the 'top down' safety argument iteratively, as the design of an item evolves. For example, the safety assessor can review the safety goal rationale argument in Fig. 1 before the functional safety or technical safety concepts have been developed, rather than waiting until the later lifecycle stages. This helps to identify weaknesses in the eventual safety argument earlier on in the project lifecycle, reducing the cost and effort resulting from any subsequent rework.

Finally, the automotive industry like many other domains is driven by tight margins and time constraints. Once a project is underway, momentum increases quickly. It is therefore essential that the visibility of the project's technical and assurance attributes and any infringements identified early so that undesired consequences are addressed. This leads to the conclusion that the review and assessment of the safety case at key product gateways will not only keep focus on the emergence of the project's and product's safety attributes, but is more likely to have a safety case at the final functional safety assessment that is legible and more readily analysable.

5 Concluding Remarks

Safety case development is a relatively new concept for many safety practitioners in the automotive industry. The timely generation of well-focussed safety cases is capable of bringing considerable benefit in the context of development and assessment, and thus of contributing to the safety assurance of automotive E/E systems. Our experience to date suggests that the primary focus of many documented safety cases for ISO 26262-compliant systems and components remains on processes. In extreme cases, this can result in bulky documentation that does little more than render explicit the standard's implicit arguments or, even, recapitulate its requirements in a different form. Broadly, we perceive an educational challenge to exist in this area even among automotive safety engineers with considerable experience in other areas.

Other characteristics have reduced the effectiveness of certain safety cases produced to meet the requirements of ISO 26262. These include: lack of focus and brevity; unnecessary repetition of information available elsewhere; and the use of inappropriate means of expression (e.g. use of GSN where a table might be more effective and vice versa). Similarly, safety cases in the automotive industry are as susceptible as those in other industries to deficiencies such as fallacies and failures to acknowledge limitations. These weaknesses are found in safety cases in other industries but, we believe, may best be countered by didactic material that is targeted specifically at the automotive industry in order to improve outreach.

References

1. ISO: ISO 26262 Road Vehicles– Functional Safety. ISO Standard (2011)
2. Graydon, P., Habli, I., Hawkins, R., Kelly, T., Knight: Arguing conformance. IEEE Software 29(3) (2012)
3. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Proc. 6th Safety-critical Sys. Symp. (1998)
4. Kelly, T.: A systematic approach to safety case management. In: Proc. Society of Automotive Engineers (SAE) World Congress (2004)
5. The Health Foundation, Using Safety Cases in Industry and Healthcare (2012) ISBN: 978-1-906461-43-0
6. Dittel, T., Aryus, H.-J.: How to “Survive” a safety case according to ISO 26262. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 97–111. Springer, Heidelberg (2010)
7. Palin, R., Habli, I.: Assurance of automotive safety – A safety case approach. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 82–96. Springer, Heidelberg (2010)
8. Habli, I., Kelly, I.: Process and product certification arguments: getting the balance right. SIGBED Review 3(4) (2006)
9. Langari, Z., Maibaum, T.: Safety cases: a review of challenges. In: International Workshop on Assurance Cases for Software-intensive Systems (ASSURE 2013), San Francisco (2013)
10. Denney, E., Pai, G., Habli, I.: Towards measurement of confidence in safety cases. In: Proc. 5th Intl. Symp. on Empirical Soft. Eng. and Measurement, pp. 380–383 (September 2011)
11. Ayoub, A., Kim, B., Lee, I., Sokolsky, O.: A systematic approach to justifying sufficient confidence in software safety arguments. In: Ortmeier, F., Lipaczewski, M. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 305–316. Springer, Heidelberg (2012)
12. Goal Structuring Notation Working Group: GSN Community Standard Version 1 (2011)

Structuring Safety Requirements in ISO 26262 Using Contract Theory

Jonas Westman¹, Mattias Nyberg², and Martin Törngren¹

¹ Royal Institute of Technology (KTH), Stockholm, Sweden
jowestm@kth.se

² Scania, Södertälje, Sweden

Abstract. ISO 26262 - "*Road vehicles-Functional Safety*" is a standard for the automotive industry, administered in an attempt to prevent potential accidents due to systematic and random failures in the Electrical/Electronic-system. ISO 26262 is based on the principle of relying on safety requirements as the main source of information to enforce correctness of design. We show that the contract theory from the SPEEDS FP6 project provides a suitable foundation to structure safety requirements in ISO 26262. Contracts provide the necessary support to separate the responsibilities between a system and its environment by explicitly imposing requirements on the environment as assumptions, in order to guarantee the safety requirements. We show this by characterizing two levels of safety requirements with contracts for an industrial system where we also show how contract theory supports the verification of consistency and completeness of safety requirements.

1 Introduction

The standard ISO 26262- "*Road vehicles-Functional Safety*" [1] is, in essence, a domain-specific systems engineering approach with a focus on functional safety. ISO 26262 is based on the principle of relying on safety requirements as the main source of information to enforce correctness of design and implementation throughout the development process. A system and its elements are in ISO 26262 characterized by being logically and technically separated from their environment in the form of a detailed interface specification and separation of responsibilities. Although not mentioned explicitly, these principles are similar to the notion of a *contract* [2], namely: based on a well-defined system boundary, the responsibilities between an environment and a system are split into a *guarantee* that models desired properties of a system, under the influence of an *assumption*, modeling expected properties of an environment. In this paper, we explore a possibility to capitalize on this similarity, by using contracts to structure safety requirements in ISO 26262.

Out of three contributions, the first contribution is that we show that the theory of contracts can enrich safety specifications as it provides the necessary support to separate the responsibilities between a system and its environment by explicitly imposing requirements on the environment as *assumptions*, in order to

guarantee the safety requirements. We show this by using the theory of contracts from the SPEEDS FP6 project¹ to characterize two levels of safety requirements for a real industrial system. Secondly, we show that the theory of contracts provides a foundation to argue for, and verify properties of safety requirements such as consistency and completeness as required by ISO 26262. Thirdly, we show that a modification to the contract theory where assumptions can model properties of an environment that are not limited to the system boundary is needed, in order to conform with the principles in ISO 26262.

Several publications associated with the project CESAR², see e.g. [3] and [4], discuss the use of contracts with respect to requirements engineering and safety standards, including ISO 26262, in general. In [5], this connection is elaborated on and a few examples (although not automotive) are shown where contracts could in fact be useful with respect to ISO 26262. However, none of [3], [4] nor [5] apply the theory on a real industrial system and mainly hypothesize about its usefulness when developing safety-critical systems. In this paper, we go in to further depths, showing an explicit use of contracts with respect to ISO 26262 by characterizing safety requirements as contracts for an industrial application example, namely the Fuel Level Display (FLD)-system, present on all heavy trucks from the manufacturer Scania.

The link between requirements engineering and contract theory is touched upon in [6,7,8], and more notably in [9], where properties of requirements, e.g. consistency, are described in a context of contracts. However, none of [6,7,8,9] address properties of safety requirements as described in ISO 26262 and the notion of completeness is not addressed to a full extent. In this paper, we establish a more elaborate connection between requirements engineering and contract theory by showing how consistency and completeness of safety requirements in ISO 26262 can be ensured through properties of contracts.

2 Illustrative Example - The Fuel Level Display-System

In this section, we introduce the illustrative example that will be used in Sec. 3 to exemplify a case where ISO 26262 relies on contract-inspired principles, and also in Sec. 5 to characterize safety requirements in ISO 26262 as contracts.

The FLD-system provides an estimate of the fuel volume in the fuel tank to the driver along with a warning if the fuel volume drops below a predefined value. The functionality provided by the FLD-system is distributed across three Electronic Control Unit (ECU)-systems, i.e. an ECU with sensors and actuators, in the Electronic/Electrical (E/E)-system: Engine Management System (EMS), Instrument Cluster (ICL), and Coordinator (COO). The ECU-systems also interact with the fuel tank that is outside of the E/E-system. COO estimates the fuel volume in the tank by relying on the output of a Kalman filter that, in turn, relies on a signal of a sensor measuring the fuel level in the tank and an estimate of the current fuel consumption provided by EMS, as inputs. The estimated fuel

¹ <http://www.speeds.eu.com/>

² <http://www.cesarproject.eu/>

volume is sent over CAN to ICL, where it is displayed to the driver along with a warning if the fuel volume in the tank is below 10%.

A development according to ISO 26262 revolves around an item which is in [1] described as "a system that implements a function at a vehicle level". For the analysis in this paper, COO, CAN, and ICL are chosen to be the item, as shown in Fig. 1, where we also illustrate the system boundary of the item.

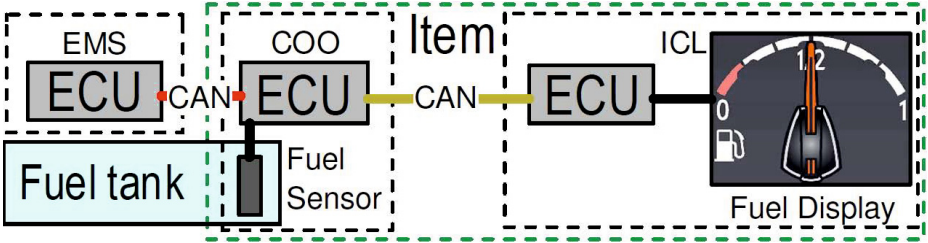


Fig. 1. System architecture of the FLD-system. The blocks represent actual ECUs, the fuel sensor, the fuel tank, and the display. The connectors represent physical cables. The borders (cross-hatched lines) represent the ECU-systems and the item.

3 Motivation - Contract-Inspired Principles in ISO 26262

In this section, we motivate our research using two rather explicit cases where ISO 26262 relies on principles similar to those of contracts from: *3-5 Item definition*; and *10-8 Safety element out of context* in [1].

Item Definition Prior to forming safety requirements, ISO 26262 first requires a description of the item as presented in *3-5 Item definition* in [1]. We consider one of its requirements and apply it to the FLD-system, i.e. requirement 5.4.2:

"The boundary of the item, its interfaces, and the assumptions concerning its interaction with other items and elements, shall be defined considering: ... d) functionality required by other items, elements and the environment; e) functionality required from other items, elements and the environment..." [1]

Concerning sub-requirement 5.4.2d); a basic functionality of the FLD-system is to provide an accurate estimation of the fuel volume to the driver, who is part of the environment. We can formalize this functionality by the requirement: *the indicated fuel volume, shown by the fuel gauge, shall not deviate more than ± 5 percent from the actual fuel volume in the tank.* We hence consider a deviation within ± 5 as acceptably accurate. However, in order for the item to be able to *guarantee* this functionality, the item needs information regarding the fuel consumption, which is provided by EMS, external to the item. Hence, concerning requirement 5.4.2e), we impose a requirement on EMS as an *assumption*: *the*

estimated fuel consumption, provided by EMS, does not deviate more than ± 1 percent from the actual fuel consumption. In conclusion, the item definition includes an interface specification with clear separation of responsibilities between the environment and the item - much like a contract.

System Element out of Context A System Element out of Context (SEooC) is a safety-related (i.e. assumed to be required to implement a safety requirement) element which is developed in isolation, that is, without the context of a specific item. Therefore, *assumptions* are made on the context of a SEooC, in the form of requirements that are likely to be allocated to its environment. The difference between a regular element (part of an item) and a SEooC is that a SEooC makes assumptions on a general environment while an element is to be integrated in a specific environment. The concept of SEooC addresses the need of subcontracting - an important aspect since companies in the automotive industry tend to rely on sub-systems developed external to the company. The concept of SEooC is similar to a description of *contracts* in a context of a distributed systems development environment where each supplier is given a design task in the form of a *guarantee*, subject to constraints under the responsibility of other actors of the company/supplier chain that are offered to this supplier as *assumptions* [2].

4 The Contract Theory of SPEEDS

The original use of contracts [10] as a pair of pre- and post-conditions as state predicates [11,12] has been extended from software to e.g. Component-Based Design and hardware [13,14]. In this paper, we choose to apply the contract theory of SPEEDS. The reason for this is that ISO 26262 is centered on the development of E/E-Systems, which encompasses both hardware and software. In the contract theory of SPEEDS, contracts are formed for Heterogeneous Rich Components (HRCs) [15], which can represent entities of software, hardware, mechanical, etc. while the other approaches are typically used only in software.

In the following sections, we will hence present the theory of contracts as described in [2],[6], [9],[8] and [7] with inspiration from [16], [17], [18], and [19]. The intent is to present the theory in accordance with these papers; however, there might be slight deviations from the original papers since only a subset deemed relevant for the present paper is presented.

The presented theory of contracts will be used in Sec. 5.1 and 5.2 to model the architecture of the FLD-system and characterize safety requirements as contracts, and then in Sec. 5.4 to support the verification of requirements properties.

4.1 Assertions and Runs

Let $P = (x_1, \dots, x_{N_P})$ be an ordered set of variables where each variable is a function of time. Consider a trajectory of values assigned to a variable x_i in P over a whole time window. A tuple of such trajectories, one for each variable in P , is called a *run* for P . An *assertion* B over P is a set of runs for P . These notions correspond to similar definitions in [2],[6], [9] and [7].

Dissimilar Sets of Variables. Given an assertion B over P' , and another set $P \subseteq P'$, the *projection* of B onto P , written $proj_{P',P}(B)$, is the set of runs obtained when each run in B is restricted to the set of variables P . Using notion of relational algebra [20] we have $proj_P(B) = \pi_P(B)$.

Given an assertion B over P' , and another set $P \supseteq P'$, the *inverse projection* of B onto P , written $proj_P^{-1}(B)$, is the set of runs obtained when each run in B is extended with all possible runs for $P \setminus P'$. We can also express this as that the projection of all runs in $proj_P^{-1}(B)$ onto P' must be in B , i.e. $proj_{P'}^{-1}(B) = \{x_P | x_P \text{ is a run for } P, proj_{P'}(\{x_P\}) \subseteq B\}$.

Receptiveness of Assertions. Let Ω_P be the set of all possible runs for P . An assertion B is said to be P -*receptive* if $proj_P(B) = \Omega_P$. This corresponds to [2],[6], and [7] where the notion of receptiveness is described as the ability of an assertion to accept any history of values offered to a subset of its ports.

4.2 Components and Contracts

A contract C modeled over a set of ordered variables P is a pair of assertions (A, G) where both A , the *assumption*, and G , the *guarantee*, are assertions over P . In practice, it can be useful to split a guarantee G or an assumption A into separate 'sub-assertions' $G.1, \dots, G.N$ or $A.1, \dots, A.N$, respectively. Sub-assertions, e.g. $G.1, \dots, G.N$, are combined by intersection to form an assertion, e.g. $G = \bigcap_{i=1}^N G.i$.

An *implementation* M , sometimes also called a design, modeled over a set of ordered variables P' , is a pair (P', B_M) where $B_M \neq \emptyset$ is an assertion over P' .

A component I is a tuple $(\mathcal{P}, \mathcal{M}_{tot}, \mathcal{C}_{tot}, \mathcal{I}_{sub})$, where

- \mathcal{P} is a pair (P_u, P_c) where P_u and P_c are non-empty mutually disjoint ordered sets of variables;
- \mathcal{M}_{tot} is a set of P_u -receptive implementations $\{M_1, \dots, M_{N_I}\}$ where each M_i is modeled over $P_u \cup P_c$;
- \mathcal{C}_{tot} is a set of contracts $\{C_1, \dots, C_{N_C}\}$ where each C_j is modeled over $P_u \cup P_c$; and
- \mathcal{I}_{sub} is a set of components $\{I_1, \dots, I_{N_{sub}}\}$ where each component $I_k = (P^k, \mathcal{M}_{tot}^k, \mathcal{C}_{tot}^k, \mathcal{I}_{sub}^k)$ is a *sub-component* of I and where $P_c^k \in \mathcal{P}^k$ of each sub-component and P_u are mutually disjoint.

Any variable in $P_u \cup P_c$ is called a *port* of I . In addition, a port in P_u is called *uncontrolled* and a port in P_c is called *controlled*. As mentioned in [2], uncontrolled/controlled ports correspond to the typical classification³ of input/output ports, respectively.

³ The ports are explicitly partitioned into uncontrolled and controlled ports in the component instead of relying on *profiles* to partition the ports of implementations, contracts, and components as in [2], [6] and [7], although the same principles apply.

Properties of Contracts. A contract $C = (A, G)$ of a component with ports $P_u \cup P_c$ is said to be *port-compatible* if A is P_c -receptive and *port-consistent* if G is P_u -receptive as described in [2,6,7], but where we add the prefix "port-" to avoid ambiguity in terminology with respect to ISO 26262 (see Sec. 5.4).

For an implementation $M = (P, B_M)$ and a contract $C = (A, G)$ modeled over the same ports, M is said to *satisfy* C , written $M \models C$, if

$$A \cap B_M \subseteq G. \quad (1)$$

In accordance with [8], given a port-compatible and -consistent contract $C = (A, G)$ of a component I with sub-components $I_1, \dots, I_{N_{sub}}$ where there exists a port-compatible and -consistent contract $C_k = (A_k, G_k)$ for each I_k , C is said to *dominate* $\{C_1, \dots, C_N\}$, if

$$A \cap \left(\bigcap_{j=1, j \neq k}^{N_{sub}} G_j \right) \subseteq A_k \text{ for } k = 1, \dots, N_{sub} \quad (2) \quad A \cap \left(\bigcap_{k=1}^{N_{sub}} G_k \right) \subseteq G \quad (3)$$

where the assertions are extended to a common set of variables, as described in Sec. 4.1, prior to applying set-theoretic operations (e.g. \cap) or comparing assertions with relations (e.g. \subseteq).

5 Structuring Safety Requirements in ISO 26262 Using Contract Theory

In order to specify contracts for an item and its elements, we first need to model an item and its environment as components as presented in Sec. 4.2. That is, using the FLD-system as an example, we model the item and its environment as two tuples $(\mathcal{P}^{item}, \mathcal{M}_{tot}^{item}, \mathcal{C}_{tot}^{item}, \mathcal{I}_{sub}^{item})$ and $(\mathcal{P}^{env}, \mathcal{M}_{tot}^{env}, \mathcal{C}_{tot}^{env}, \mathcal{I}_{sub}^{env})$, respectively where COO, CAN and ICL are elements in \mathcal{I}_{sub}^{item} , and fuel tank and EMS are elements in \mathcal{I}_{sub}^{env} , as shown in Fig. 2. We only model variables as ports to the components if we need to refer to them in an assumption or guarantee. For example, the indicated fuel volume, shown by the fuel gauge, is modeled as the controlled port `indicatedFuelVolume [%]` of the item.

We assume that the item only implements the functionality of the FLD-system. In reality, the ECU-systems (COO and ICL) also implement other functionalities; e.g. COO also implements Cruise Control. We also assume that a more general contract is already in place concerning power delivery to the ECU-systems and we therefore model the status of the ignition (`ignition [Bool]`) as a port on all ECU-systems.

5.1 Characterizing Safety Goals as a Contract

In this section, we show how we can characterize Safety Goals (SGs), i.e. top-level safety requirements, in ISO 26262 using contracts, by specifying safety goals for an item as a guarantee, given explicit requirements on its environment, expressed by an assumption. To illustrate these principles, we specify a contract $C_{item} \in$

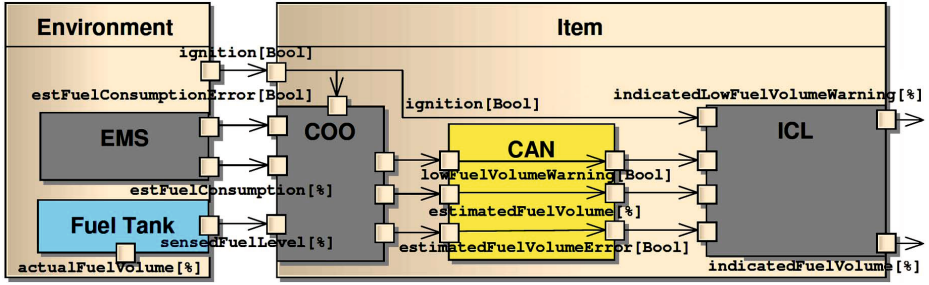


Fig. 2. A representation of the item (COO, CAN, and ICL) and the environment (Fuel Tank and EMS) modeled as components using a SysML internal block-diagram. Inputs and outputs to the blocks correspond to uncontrolled and controlled ports, respectively. Since we assumed that CAN delivers its signals immediately and with perfect accuracy (see Sec. 5.2), we choose to model signals over CAN as one variable.

C_{tot}^{item} for the FLD-system in Table 1 where the guarantee G_{item} is a Safety Goal, and the assumption A_{item} expresses requirements on the environment of the item. The Safety Goal G_{item} can be interpreted as: *the FLD-system shall not provide misleading information to the driver while driving*. The assumption A_{item} can be interpreted as: *the fuel sensor shall be correctly installed and that the EMS shall provide an accurate estimate of the fuel consumption*. The contract C_{item} therefore imposes, through A_{item} , its requirements on the environment in order to achieve its Safety Goal. That is, if e.g. the fuel sensor is installed incorrectly, the item cannot guarantee the Safety Goal G_{item} .

In Table 1, we formalize the notion of driving as a state of the vehicle when the fuel volume derivative is less than zero. We further let the Safety Goal G_{FLDS}^{item} be characterized by three safe states (while driving): the `indicatedFuelVolume[%]`, shown by the fuel gauge, is less than `actualFuelVolume[%]`; the warning `lowFuelVolumeWarning[Bool]` is functioning correctly; and the `indicatedFuelVolume[%]` is less than 0% (degraded state).

The assumption A_{item} in Table 1 is split up into three sub-assumptions where we assume a relation between `actualFuelVolume[%]` and `sensedFuelLevel[%]` from the fuel sensor, and between the derivative of `actualFuelVolume[%]` and the estimated fuel consumption (`estFuelConsumption[%]`) and its signal status (`estFuelConsumptionError[%]`), respectively. The assumed relation between `ignition[Bool]` and the fuel volume derivative may seem redundant; it is, however, motivated in Sec. 5.4.

5.2 Characterizing Functional Safety Requirements as Contracts

In the same manner in which we characterized a Safety Goal as a contract in Sec. 5.1, we can characterize FSRs as contracts for the elements of an item. Applying this concept to the FLD-system results in three contracts C_{COO} , and C_{ICL} , and C_{CAN} , with FSRs as guarantees, given assumptions on their environment.

Table 1. Contract of the item, characterizing the Safety Goal for the FLD-system

<i>A_{item}</i>	
1	If the derivative of <code>actualFuelVolume[%]</code> is less than 0, then <code>ignition[Bool]</code> is on (true).
2	If the derivative of <code>actualFuelVolume[%]</code> is less than or equal to 0, then <code>sensedFuelLevel[%]</code> shall not deviate more than $\pm 10\%$ from <code>actualFuelVolume[%]</code> in the fuel tank.
3	If the derivative of <code>actualFuelVolume[%]</code> is less than or equal to 0, then <code>estFuelConsumption[litres/h]</code> shall not deviate more than $\pm 1\%$ from the derivative of <code>actualFuelVolume[%]</code> ; or <code>estFuelConsumptionError[Bool]</code> shall be set to true.
<i>G_{item}</i> (Safety Goal)	
1	If the derivative of <code>actualFuelVolume[%]</code> is less than 0, <code>indicatedFuelVolume[%]</code> , shown by the fuel gauge, shall be less than <code>actualFuelVolume[%]</code> ; or <code>indicatedLowFuelLevelWarning[Bool]</code> shall be active (true) when the <code>actualFuelVolume[%]</code> is below 10%; or <code>indicatedFuelVolume[%]</code> , shown by the fuel gauge, shall show a value below 0%.

The contracts C_{COO} and C_{ICL} are shown in Tables 2 and 3, respectively. The contract C_{CAN} is not included due to space restrictions, but can be found in [21].

The FSR G_{COO} in Table 2 expresses that COO shall: provide an estimate of the fuel volume (`estimatedFuelVolume[%]`) that is less than or equal to `actualFuelVolume[%]` and a Boolean signal (`lowFuelVolumeWarning[Bool]`) indicating if `actualFuelVolume[%]` is below 10%; or set the signal status of the estimated fuel volume (`estimatedFuelVolumeError[Bool]`) to erroneous (true). Note that assumption A_{COO} in Table 2 is identical to A_{item} in Table 1.

Table 2. Contract for COO, characterizing a FSR for the FLD-system

<i>A_{COO}</i>	
1	If the derivative of <code>actualFuelVolume[%]</code> is less than 0, then <code>ignition[Bool]</code> is on (true).
2	If the derivative of <code>actualFuelVolume[%]</code> is less than or equal to 0, then <code>sensedFuelLevel[%]</code> shall not deviate more than $\pm 10\%$ from <code>actualFuelVolume[%]</code> in the fuel tank.
3	If the derivative of <code>actualFuelVolume[%]</code> is less than or equal to 0, then <code>estFuelConsumption[litres/h]</code> shall not deviate more than $\pm 1\%$ from the derivative of <code>actualFuelVolume[%]</code> ; or <code>estFuelConsumptionError[Bool]</code> shall be set to true.
<i>G_{COO}</i>	
1	If the derivative of <code>actualFuelVolume[%]</code> is less than 0, then <code>estimatedFuelVolume[%]</code> shall be less than <code>actualFuelVolume[%]</code> and if <code>actualFuelVolume[%]</code> is below 10%, then <code>lowFuelVolumeWarning[Bool]</code> shall be active (true); or <code>estimatedFuelVolumeError[Bool]</code> is set to true

Table 3. Contract for ICL, characterizing FSRs for the FLD-system

<i>A_{ICL}</i>	
1	<code>estimatedFuelVolume[%]</code> , including its signal status <code>estimatedFuelVolumeError[Bool]</code> , sent from COO is immediately received by ICL without loss of accuracy.
2	<code>lowFuelVolumeWarning[Bool]</code> sent from COO is immediately received by ICL without loss of accuracy.
<i>G_{ICL}</i>	
1	If <code>ignition[Bool]</code> is on (true) and <code>estimatedFuelVolumeError[Bool]</code> is false, then <code>indicatedFuelVolume[%]</code> , shown by the fuel gauge, shall correspond to the <code>estimatedFuelVolume[%]</code> ; or <code>indicatedFuelVolumeWarning[Bool]</code> shall correspond to <code>lowFuelVolumeWarning[Bool]</code> ;
2	If <code>ignition[Bool]</code> is on and <code>estimatedFuelVolumeError[Bool]</code> is true, then <code>indicatedFuelVolume[%]</code> shall show a value below 0%.

As presented in Table 3, ICL basically acts like an actuator without any substantial additional logic. It assumes that the CAN-signals sent from COO are delivered immediately with perfect accuracy, as modeled by A_{ICL} . When `ignition[Bool]` is on, G_{ICL} expresses that the fuel gauge shall display `estimatedFuelVolume[%]` as `indicatedFuelVolume[%]` and `lowFuelVolumeWarning[Bool]` as `indicatedLowFuelVolumeWarning[Bool]` in case the signal status of the estimated fuel volume is valid (`estimatedFuelVolumeError[Bool] = false`). In case it is erroneous, the fuel gauge shall indicate a value below 0%.

The set of FSRs G_{CAN} where each FSR is a sub-guarantee, expresses that all CAN-signals are delivered immediately and with perfect accuracy. The guarantee G_{CAN} is hence equal to the assumption A_{ICL} in Table 3. This is of course not realistic, but for this system, safety aspects are not highly affected due to slight delays over CAN and such a simplification is therefore deemed to be justifiable. CAN does not impose any requirements on its environment and the assumption A_{CAN} is thus receptive to its input ports (see Sec. 4.1).

5.3 Modification of the Contract Theory of SPEEDS and Its Implications

In Table 1 and 2, it can be noted that the assumptions A_{item} and A_{COO} , and guarantees G_{item} and G_{COO} are not limited to the system boundary, i.e. to the ports, of the item and COO, respectively. This is necessary since ISO 26262 requires that properties of an environment, not limited to the system boundary of the item/element, are taken into consideration, see e.g. requirement 5.4.2e) from 3-5 *Item definition* in Sec. 3. Hence, the limitation that a contract must be modeled over the ports of its component (see Sec. 4.2) has been relaxed.

As a result of this, using the requirements on the low fuel volume warning in Table 1 as an example, we are able to express the (sub-)requirement in G_{item} that *the warning shall be active when the actual fuel volume is below 10%*, given the assumption A_{item} that *EMS shall provide an accurate estimation of the fuel consumption and that the fuel sensor has been installed correctly*. If we, in contrast, restrict contracts to be modeled over the ports of its component, it is impossible to express the assumption A_{item} and the Safety Goal G_{item} , since the actual fuel volume is not a port of the item, see Fig. 2. In the case of A_{item} , for example, we cannot express that there is in fact a relation between the signal provided by the fuel sensor and the actual fuel volume.

This modification has a slight impact on the properties of contracts as presented in Sec. 4.2. The relation in (1) is generalized in the sense that the constraint that B_M , A , and G must be modeled over the same ports is removed. An implementation B_M and the assertions A and G must therefore be extended to a common set of variables before applying intersection and comparing with the subset relation (see Sec. 4.1). We let P_Ω be the universal set of all ports and P_{intP} the set of all ports of all sub-components I_i of I and of all sub-components of each I_i , and so forth. We say that a contract $C = (A, G)$ of a component I is port-consistent if G is $P_\Omega \setminus P_C$ -receptive and port-compatible if A is $P_C \cup (P_{intP} \setminus P_u)$ -receptive.

5.4 Verifying Consistency and Completeness of Safety Requirements

ISO 26262 is based on the principle of relying on requirements as the main source of information to enforce *correctness* of design and implementation throughout the development process. This amounts to verifying properties of requirements and of sets of requirements as mentioned in 8-6 *The specification and management of safety requirements* in [1]. One of these properties is *consistency* of requirements, which means that "an individual requirement does not contradict itself" (*internal consistency*) and that "a set of requirements do not contradict each other"[1] (*external consistency*). Another property is *completeness*, which means that "the safety requirements at one requirement level fully implement all safety requirements of the previous level" [1].

We show, in the following, that the contract theory of SPEEDS supports the verification of consistency and completeness of safety requirements - as required by ISO 26262. As indicated in Sec. 5.1 and 5.2, we consider a safety requirement or a set of safety requirements as a guarantee G of a contract C for an element/item, where A expresses the requirements on its environment. With inspiration from [9], we consider a safety requirement G of a contract C , to be *internally consistent* if $G \neq \emptyset$, i.e. if there exists at least one run in G . As indicated in Theorem 1, internal consistency of a safety requirement G can be ensured through a port-consistent contract (A, G) (see [21] for proof).

Theorem 1. *If a contract $C = (A, G)$ of a component I is port-consistent (see Sec. 4.2 and 5.3), then the safety requirement G , is internally consistent.*

The dominance property of contracts in Sec. 4.2 can be used to support the verification of completeness and external consistency as indicated in Theorem 2 (see [21] for proof). Since we consider the use of safety requirements in a context of contracts, external consistency does not only amount to showing that a set of safety requirements $\{G_1, \dots, G_N\}$ are not contradictory, but also not contradictory with respect to their corresponding assumptions $\{A_1, \dots, A_N\}$. We thus consider a set of safety requirements $\{G_1, \dots, G_N\}$ with corresponding assumptions $\{A_1, \dots, A_N\}$ where $A_i, G_i \in$ a contract C_i to be *externally consistent* if $(\bigcap_{i=1}^{N_{sub}} A_i) \cap \bigcap_{i=1}^{N_{sub}} G_i \neq \emptyset$.

We consider completeness for a scenario where we have a contract $C = (A, G)$ of a component I with sub-components $I_1, \dots, I_{N_{sub}}$ and there exists a contract $C_k = (A_k, G_k)$ for each I_k . We say that *a set of safety requirements $\{G_1, \dots, G_N\}$ is complete with respect to G* if for any implementation M_k of each I_k that satisfies its contract, i.e. if $M_k \models C_k$, then $A \cap (\bigcap_{k=1}^{N_{sub}} M_k) \subseteq G$.

Theorem 2. *Let $C = (A, G)$, be a port-compatible and -consistent contract of a component I with sub-components $I_1, \dots, I_{N_{sub}}$, and $C_k = (A_k, G_k)$ be a port-compatible and -consistent contract of each I_k . Furthermore, if G and $\{G_1, \dots, G_{N_{sub}}\}$ are safety requirements, then $\{G_1, \dots, G_{N_{sub}}\}$ are externally consistent and complete with respect to G if C dominates $\{C_1, \dots, C_N\}$.*

If we investigate the FLD-system example, we see that the contract C_{CAN} (see [21]) and all contracts in Tables 1, 2, and 3 are port-consistent and -compatible since they respect the constraints in Sec. 4.2 and 5.3, see [21] for further clarification. Internal consistency is hence ensured through Theorem 1. We can verify the external consistency and completeness of the set of FSRs G_{CAN} (see [21]), G_{COO} , and G_{ICL} (See Tables 2 and 3) with respect to the Safety Goal G_{Item} in Table 1 through Theorem 2, if we can show that:

$$A_{Item} \cap \left(\bigcap_{j \neq i} G_j \right) \subseteq A_i \text{ for } i = COO, CAN, ICL \text{ and} \quad (4)$$

$$A_{Item} \cap (G_{COO} \cap G_{CAN} \cap G_{ICL}) \subseteq G_{Item}. \quad (5)$$

Relation (4) is trivially true, since $A_{Item} = A_{COO}$, $G_{CAN} = A_{ICL}$, and A_{CAN} is receptive to the set of universal ports P_Ω . Concerning relation (5), since we have assumed that the fuel volume derivative is always negative when `ignition[Bool]` is on (see A_{Item} in Table 1), we can conclude that when `indicatedFuelVolume[%]` is less than zero and `ignition[Bool]` is on (see G_{ICL} in Table 3) corresponds to a safe state of the item, see G_{Item} in Table 1. If `estimatedFuelVolumeError[%]=false` and `ignition[Bool]` is on, ICL will either display the fuel warning (`indicatedLowFuelVolumeWarning[Bool]`) in case `actualFuelVolume[%] < 10%` or `indicatedFuelVolume[%]` that is less than `actualFuelVolume[%]`, since this is guaranteed by the contracts of CAN and COO (see [21] and Table 2). These states both correspond to safe states of the item and hence relation (5) is also true. Through Theorem 2, we can hence claim that the FSRs, expressed by G_{COO} , G_{CAN} , and G_{ICL} , are externally consistent and complete with respect to the Safety Goal, expressed by G_{item} .

6 Conclusions

We have shown in Sec. 5.1 and 5.2 that safety requirements can be characterized as contracts for an item and its elements, with guarantees that constitute the safety requirements, given explicit requirements on their environments as assumptions. A Contract therefore enriches a safety specification for an item/element by explicitly declaring what each element/item expects from the environment to ensure that the safety requirements are satisfied.

We have also shown in Sec. 5.4 that consistency and completeness of safety requirements can be ensured through verifying the dominance property of contracts.

However, these achievements were only made possible due to a modification of the contract theory of SPEEDS, as presented in Sec. 5.3. The modification relaxes the constraint that a contract must be modeled over the ports of its component.

We hence conclude that the principles of contracts provide a suitable foundation to structure safety requirements in ISO 26262.

References

1. ISO: 26262 - Road vehicles-Functional safety (2011)
2. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple Viewpoint Contract-Based Specification and Design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
3. Blanquart, J.-P., et al.: Towards Cross-Domains Model-Based Safety Process, Methods and Tools for Critical Embedded Systems: The CESAR Approach. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 57–70. Springer, Heidelberg (2011)
4. Baumgart, A., Reinkemeier, P., Rettberg, A., Stierand, I., Thaden, E., Weber, R.: A Model-Based Design Methodology with Contracts to Enhance the Development Process of Safety-Critical Systems. In: Min, S.L., Pettit, R., Puschner, P., Ungerer, T. (eds.) SEUS 2010. LNCS, vol. 6399, pp. 59–70. Springer, Heidelberg (2010)
5. Damm, W., Josko, B., Peinkamp, T.: Contract Based ISO CD 26262 Safety Analysis. In: Safety-Critical Systems, SAE (2009)
6. Sangiovanni-Vincentelli, A.L., Damm, W., Passerone, R.: Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *Eur. J. Control* 18(3), 217–238 (2012)
7. Benveniste, A., Caillaud, B., Passerone, R.: Multi-Viewpoint State Machines for Rich Component Models (2008)
8. Graf, S., Quinton, S.: Contracts for BIP: Hierarchical Interaction Models for Compositional verification (2007)
9. Benveniste, A., et al.: Contracts for the Design of Embedded Systems. Part II: Theory (March 2013), <http://www.irisa.fr/distribcom/benveniste/pub/>
10. Meyer, B.: Applying “Design by Contract”. *IEEE Computer* 25, 40–51 (1992)
11. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. *Commun. ACM* 12(10), 576–580 (1969)
12. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18(8), 453–457 (1975)
13. Giese, H.: Contract-based Component System Design. In: Thirty-Third Annual Hawaii Int. Conf. on System Sciences (HICSS-33). IEEE Press, Maui (2000)
14. Sun, X., et al.: Contract-based System-Level Composition of Analog Circuits. In: 46th ACM/IEEE Design Automation Conf., DAC 2009, pp. 605–610 (July 2009)
15. Damm, W.: Controlling Speculative Design Processes Using Rich Component Models. In: Fifth International Conference on Application of Concurrency to System Design, ACSD 2005, pp. 118–119 (June 2005)
16. Back, R.-J., Wright, J.V.: Contracts, Games and Refinement. In: Information and Computation, p. 200–0. Elsevier (1997)
17. Alfaro, L.D., Henzinger, T.A.: Interface Theories for Component-based Design, pp. 148–165. Springer (2001)
18. Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. In: Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, pp. 51–65. MIT Press, Cambridge (1988)
19. Negulescu, R.: Process spaces. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 199–213. Springer, Heidelberg (2000)
20. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13(6), 377–387 (1970)
21. Westman, J., Nyberg, M., Törngren, M.: Structuring Safety Requirements in ISO 26262 using Contract Theory. Technical Report TRITA MMK 2013:04, KTH (March 2012)

Flexible Unequal Error Control Codes with Selectable Error Detection and Correction Levels

Luis-J. Saiz-Adalid, Pedro-J. Gil-Vicente, Juan-Carlos Ruiz-García,
Daniel Gil-Tomás, J.-Carlos Baraza, and Joaquín Gracia-Morán

Instituto ITACA – Universitat Politècnica de València
Camino de Vera, s/n, 46022, Valencia, Spain

{ljsaiz,pgil,jcruizg,dgil,jcbaraza,jgracia}@itaca.upv.es

Abstract. Unequal Error Control (UEC) codes provide means for handling errors where the codeword digits may be exposed to different error rates, like in two-dimensional optical storage media, or VLSI circuits affected by intermittent faults or different noise sources. However, existing UEC codes are quite rigid in their definition. They split codewords in only two areas, applying different (but limited) error correction functions in each area. This paper introduces Flexible UEC (FUEC) codes, which can divide codewords into any required number of areas, establishing for each one the adequate error detection and/or correction levels. At design time, an algorithm automates the code generation process. Among all the codes meeting the requirements, different selection criteria can be applied. The code generated is implemented using simple logic operations, allowing fast encoding and decoding. Reported examples show their feasibility and potentials.

Keywords: error detection and correction codes, information redundancy, unequal error control codes.

1 Introduction

Error correction codes (ECCs) are widely used in today's computer systems to provide reliable delivery and storage of digital data over unreliable communication channels and memories. Most ECCs are based on the premise that all bits in a codeword require the same error control level. However, this vision of the problem is not valid for application domains where the bit error rate (BER) does not homogeneously affect to all codeword bits [1]. Far from being marginal, the problem of variable BER (vBER) is getting more and more important. For instance, intermittent faults in VLSI circuits appear repeatedly and non-deterministically in the same place. They only increase the BER of the affected locations, but not the BER of the rest of the bits in the word. Although the specific causes leading to this trend are out the scope of this paper, it is worth noting that, nowadays, 6.2% of errors in memory subsystems [2], and 39% of hardware errors in microprocessors reported to operating systems have an intermittent nature [3]. vBER problem also applies to Volume Holographic Memories (VHM) and other two dimensional optical storage, where the BER of readout data

from the edge is much higher than that from the center of the media [4]. This threat must be taken into account as this technology for data-storage hierarchy presents short access time, high aggregate data-transfer rate, and large storage capacity.

The aforesaid examples motivate the increasing interest that asymmetric error control has in current, and will have in future, computer-based systems. Unequal error control (UEC) codes [1] establish different error control levels in diverse codeword areas. In practice, existing UEC codes simply split codewords in two parts. They apply either full error correction (i.e. fixing all the possible errors affecting that word area) [1] or burst error correction (i.e. fixing burst errors of a given length affecting that area) [5] in the strongly controlled area. In contrast, only single error correction (and sometimes double error detection) is applied in the weakly controlled area.

The limitation of using full error correction in the strongly controlled area is the high level of redundancy required. In contrast, burst error correction, although less redundancy demanding, does not cover some classical error patterns, such as double random errors (two errors separated beyond the controlled burst length). On the other hand, single error correction may be insufficient even for a weakly controlled area, as multiple errors are becoming more and more frequent in today's systems [6]. Thus, the main drawback of existing UEC codes is their lack of flexibility, as designers cannot use different error control functions on each area, according to the specifications of each system, application or context of use.

This paper proposes Flexible UEC (FUEC) codes, a new type of UEC codes enhanced for flexibility. They allow to establish any desired number of control areas in a codeword, in relation to the needs, and to deploy the adequate error control strategy in each part. The focus will be placed on how to combine single, multiple and burst error correction and/or detection capabilities, and selectively apply them in each identified codeword area. In case of multiple solutions, different selection criteria can be used.

The rest of this paper is organized as follows. Section 2 introduces the very basic notions related to ECCs and UEC codes, while Section 3 details the methodology to generate FUEC codes. In Section 4, the feasibility and potentials of FUEC codes are discussed. Finally, Section 5 provides some conclusions and ideas for future work.

2 UEC Codes: Background

UEC codes are a type of linear block codes, a kind of ECCs. For a better understanding of their potentials, this section introduces some important notions. Basics will be first applied to ECCs and then extended to UEC codes.

2.1 Basics on Encoding and Decoding

An (n, k) binary ECC encodes a k -bit input word in an n -bit output word [7]. Fig. 1 synthesizes the encoding and decoding processes. The input word $\mathbf{u}=(u_0, u_1, \dots, u_{k-1})$ is a k -bit vector which represents the original data. The codeword $\mathbf{b}=(b_0, b_1, \dots, b_{n-1})$ is a vector of n bits, where the code adds the required redundancy. It is transmitted across the channel which delivers the received word $\mathbf{r}=(r_0, r_1, \dots, r_{n-1})$. The error

vector $\mathbf{e}=(e_0, e_1, \dots, e_{n-1})$ models the error induced by the channel. If no error has occurred in the i th bit, $e_i=0$; otherwise, $e_i=1$. In this way, \mathbf{r} can be interpreted as $\mathbf{r}=\mathbf{b}\oplus\mathbf{e}$.

The parity-check matrix \mathbf{H} of a linear block code defines the code. For the encoding process, \mathbf{b} must accomplish the requirement $\mathbf{H}\cdot\mathbf{b}^T=\mathbf{0}$. For syndrome decoding, the syndrome is defined as $\mathbf{s}^T=\mathbf{H}\cdot\mathbf{r}^T$, and it exclusively depends on \mathbf{e} :

$$\mathbf{s}^T = \mathbf{H}\cdot\mathbf{r}^T = \mathbf{H}\cdot(\mathbf{b} \oplus \mathbf{e})^T = \mathbf{H}\cdot\mathbf{b}^T \oplus \mathbf{H}\cdot\mathbf{e}^T = \mathbf{H}\cdot\mathbf{e}^T \tag{1}$$

There must be a different \mathbf{s} for each correctable \mathbf{e} . If $\mathbf{s}=\mathbf{0}$, then $\mathbf{e}=\mathbf{0}$. So, \mathbf{r} is correct. Otherwise, the syndrome decoding is performed by addressing a lookup table relating each \mathbf{s} with the decoded error vector $\hat{\mathbf{e}}$. By XORing $\hat{\mathbf{e}}$ from \mathbf{r} , the decoded codeword $\hat{\mathbf{b}}$ is obtained: $\hat{\mathbf{b}} = \mathbf{r} \oplus \hat{\mathbf{e}}$. If \mathbf{s} is non-zero, but the lookup table has no entry for that syndrome, the error is detected, but cannot be corrected.

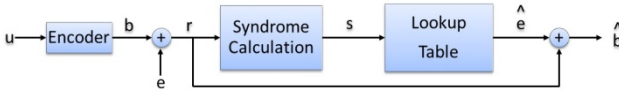


Fig. 1. Encoding, channel crossing and decoding process

Encoding and decoding functions are similar for UEC codes (although \mathbf{b} is divided in two areas and the $n-k$ redundant bits added to \mathbf{b} are differently located). Such functions and their implementation can be deduced from \mathbf{H} , as exemplified in Section 3.2. The encoding and decoding circuits can be implemented using XOR trees.

Let us focus now on the various types of errors typically addressed using ECCs and UEC codes. It must be noted that an accurate modeling of such errors is essential to reduce the level of redundancy induced in the resulting codeword, and thus the number of bits required in such a word to limit its length.

2.2 Error Models

As previously stated, the difference between \mathbf{b} and \mathbf{r} is induced by an unreliable channel that introduces \mathbf{e} . This section describes the error models used in this work.

The term **random error** refers to one or more bits in error, distributed randomly in \mathbf{b} . Random errors can be single (only one bit is affected) or multiple. Single errors are commonly produced by single event effects (SEEs) [8]: single event upsets (SEUs, in random access memories), single event transients (SETs, in combinational logic), etc.

Multiple errors are becoming more frequent as integration scales increase [6][9], although they usually manifest as burst errors, rather than randomly [10]. A **burst error** is a multiple error that spans l (named burst length) bits in a word [1], i.e. a group of adjacent bits where, at least, the first and the last bits are in error. The physical causes of a burst error are diverse [9][10]: crosstalk effects induced between neighbor wires in parallel buses, noise affecting several bits in a serial transmission, high energy cosmic ray that hits some neighbor positions in storage elements, etc.

Let us introduce now some notation about error vectors used from now on.

Let E_* be the set of all possible error vectors. Its cardinality is $|E_*| = 2^n$. The elements in this set can be grouped in different subsets. For example, let E_w be the set of error vectors with Hamming weight w (the Hamming weight is the number of 1s in a binary word). For example, if $n=3$ and $w=1$, $E_1 = \{(100), (010), (001)\}$.

Regarding burst errors, let E_{Bl} be the set of burst error vectors, where l is the burst length. For example, if $n=6$ and $l=4$, the E_{B4} set is composed by these error vectors:

$$E_{B4} = \{ (111100), (011110), (001111), (101100), (010110), (001011), \\ (110100), (011010), (001101), (100100), (010010), (001001) \}$$

Let us now consider E_+ as the set of all error vectors that determine which errors can be corrected by a given code, including the no-error subset (E_0). When using syndrome decoding, $|E_+|$ determines the minimum number of syndromes required to correct the selected errors, i.e. the condition $|E_+| \leq 2^{n-k}$ must be satisfied.

The minimum set of error vectors to be detected (excluding those in E_+) is represented by E_Δ . Nevertheless, other error vectors, not included in E_Δ , may be detected. This set is used to indicate the minimum detection requirements of a code.

When related to UEC codes, the previous definitions do not apply to the whole word, but to a subset of bits, defining a *control area*. Superscript numbers are then used to distinguish areas ($E_*^{0..7}$, for example). These error subsets will be useful to define different error control functions in distinct areas of a codeword.

2.3 Unequal Error Control vs. Unequal Error Protection Codes

UEC codes are developed considering that some bits require higher error control than others. Although they share some common features with Unequal Error Protection (UEP) codes, they must not be confused, since their definitions, objectives and applications are quite different.

UEP codes [11], typically used in multimedia and control communications, protect some digits in a codeword against a higher number of errors than others. UEP codes ensure the correct decoding of the strongly protected part, and accept the eventual incorrectly decoding of the weakly protected part, i.e. the integrity of the whole codeword is desired but not required.

Conversely, UEC codes consider distinct error control levels in a codeword, in such a way that a part of the word is more strongly controlled against errors than the rest. UEC codes enable different error control functions to be applied to distinct parts of a codeword, but error protection is applied to the whole word, thus making unacceptable the incorrect decoding of any part of such word.

Thus, from a protection level viewpoint, UEP codes are different from UEC codes. UEC codes decode correctly the word if the error requirements are met; otherwise, the whole word can be corrupted. UEP codes tolerate a wrong decoding if it is “good enough”, i.e. errors in the weakly protected bits are acceptable. So, UEP codes have different protection levels, while UEC codes protect all codeword bits equally.

From a control level standpoint, UEP codes usually consider the codeword as a whole, all bits having the same error rate (under a given number of errors, some bits are guaranteed to be correctly decoded, while others could be in error or miscorrected). UEC codes commonly consider multiple areas, and different error control levels are applied to each area (while fault hypothesis are met, errors are

covered and the whole word is correctly decoded; otherwise, the integrity of the codeword cannot be guaranteed). From this perspective, UEC codes have various control levels, while UEP codes can only have one.

The aforementioned differences obviously establish different contexts of use for UEP and UEC codes. On one hand, UEP codes are used when some information in part of the word is more important than in other parts (e.g. control information in communication messages or headers in multimedia transmission). On the other hand, UEC codes are appropriate in scenarios where constant bit error rates cannot be assumed (e.g. VHM or VLSI circuits affected by intermittent faults).

As motivated in Section 1, the increasing importance of vBER problem is making UEC codes a more and more interesting solution. Nevertheless, existing UEC codes have some limitations, mainly related to both the limited number of parts and the options for the error control functions applied on each part. This work proposes FUEC codes to solve them.

3 Flexible Unequal Error Control Codes

Existing UEC codes split codewords only in two parts, and they offer limited error control functions. Full error correction in the strongly controlled area requires a high level of redundancy, whereas burst error correction does not cover some classical error patterns. Single error correction may be insufficient even for a weakly controlled area. So, the main drawback of existing UEC codes is their lack of flexibility.

For the sake of understanding, a simple design example will be introduced to provide an overview of FUEC codes, and present their features and potentials.

3.1 Overview

Let us consider a two-dimensional optical storage with vBER [1]. In VHM, for example, the BER of readout data from the edge of the media is much higher than that from the center [4]. Asymmetric error control would reduce the redundancy-coverage ratio. Two control levels are applied in [1]; nevertheless, the BER has not only two levels in VHM actually. In fact, the BER is proportional to the distance from the center of the media [4]. Hence, more than two areas should be considered.

Let us study a simple example. For the sake of simplicity, words are 12-bit long, divided in three 4-bit areas, as presented in Fig. 2(a). This methodology can be applied to a higher number of bits, as it will be shown in Section 4.

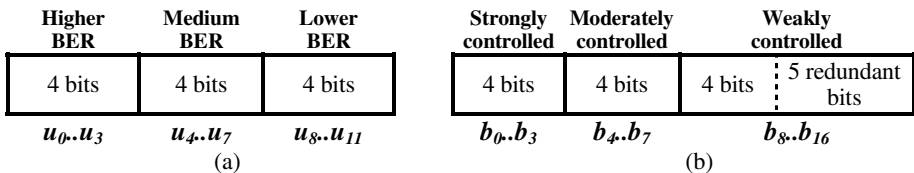


Fig. 2. Layout of the input word (a) and the encoded word (b)

The following fault hypothesis are considered: i) all 1-bit errors and all 2-bit burst errors should be corrected, and all 2-bit errors and 3-bit burst errors should be detected in the strongly controlled area (the one with higher BER); ii) all 1-bit errors and all 2-bit burst errors must be corrected in the moderately controlled area, with no additional detection; iii) 1-bit error correction and 2-bit burst error detection will be implemented in the weakly controlled area (the one with lower BER).

The value of k for the code to be designed is 12. The selection of the number of redundant bits (which finally determines the value of n) must allow finding an \mathbf{H} matrix which solves the error control requirements. It is possible to estimate the number of redundant bits depending on these requirements, as explained later. Next, the methodology used to generate FUEC codes is applied to this example.

3.2 Methodology Description

Several parameters must be set to design a code: the data length (k), the encoded word length (n), the set of error vectors to be corrected (\mathbf{E}_+) and detected (\mathbf{E}_Δ). For FUEC codes, other parameters have to be considered to define \mathbf{E}_+ and \mathbf{E}_Δ : the number of control areas of the codeword, the boundaries (that is, the first and last bits) of each area, and the error control level to apply to each area.

With these parameters, it is possible to obtain an $((n-k) \times n)$ \mathbf{H} matrix able to correct and detect the selected errors, if it exists. The methodology proposed consists on considering all possible matrices. As stated above, \mathbf{s} exclusively depends on \mathbf{e} (see (1)). \mathbf{H} must satisfy condition (2), as there must be a different syndrome for each correctable error. The condition for additional error detection is (3).

$$\mathbf{H} \cdot \mathbf{e}_i^T \neq \mathbf{H} \cdot \mathbf{e}_j^T; \forall \mathbf{e}_i, \mathbf{e}_j \in \mathbf{E}_+ | \mathbf{e}_i \neq \mathbf{e}_j \quad (2)$$

$$\mathbf{H} \cdot \mathbf{e}_i^T \neq \mathbf{H} \cdot \mathbf{e}_j^T; \forall \mathbf{e}_i \in \mathbf{E}_\Delta, \mathbf{e}_j \in \mathbf{E}_+ \quad (3)$$

That is, each detectable error must generate a syndrome which is different to all the syndromes generated by the correctable errors. However, several detectable errors may have the same syndrome.

Searching \mathbf{H} can be considered a Boolean satisfiability (SAT) problem. Previous proposals to solve this problem [10][12] are focused on specific applications. Our proposal is more general: in three successive steps, our algorithm is able to find any binary linear block code, if it exists, just selecting the set of error vectors to be corrected. So, the first step is to determine \mathbf{E}_+ and \mathbf{E}_Δ . Then, the algorithm tries to find an \mathbf{H} matrix able to solve conditions (2) and (3). Finally, as several solutions can be found, one of them can be selected using different criteria.

Determining \mathbf{E}_+ and \mathbf{E}_Δ Error Vector Sets. Taking the example presented in Section 3.1, the set of correctable errors is $\mathbf{E}_+ = \mathbf{E}_0 \cup \mathbf{E}_1 \cup \mathbf{E}_{B2}^{0..7}$. It includes the no-error vector (\mathbf{E}_0), all single bit errors (\mathbf{E}_1) and all 2-bit burst errors in the areas with higher and medium BER ($\mathbf{E}_{B2}^{0..7}$). As stated above, the condition $|\mathbf{E}_+| \leq 2^{n-k}$ must be satisfied. In this case, $|\mathbf{E}_+| = |\mathbf{E}_0| + |\mathbf{E}_1| + |\mathbf{E}_{B2}^{0..7}| = 1 + n + 7 \leq 2^{n-12}$. From this

expression, $n \geq 17$, that is, at least 5 redundant bits are required. If $n = 17$, $|E_+| = 25$ and $2^{n-k} = 32$. The layout of the encoded word is shown in Fig. 2(b), and the vectors representing the errors to be corrected are included in Table 1.

Positioning the redundant bits in the weakly controlled area is not a requirement. The methodology allows positioning them in any area, considering that the number of error vectors to be included may increase, and probably the number of redundant bits ($n-k$) too. In fact, the position of redundant bits depends on the design specifications.

Let us define now E_Δ . As decided in the requirements of this example, $E_\Delta = (E_2^{0..3} - E_{B2}^{0..3}) \cup E_{B3}^{0..3} \cup E_{B2}^{7..16}$. The seven (32-25) syndromes not used for correction are employed for the detection of these error vectors, grouped in Table 2.

This is just an example. In the same way, other sets of error vectors can be generated, depending on the requirements of the code to be designed.

Table 1. Vectors representing the errors to be corrected in the considered example

Correctable errors (E_+)	Error subset
(0000000000000000)	No error (E_0)
(1000000000000000) (0100000000000000) (0010000000000000) (0001000000000000) (0000100000000000) (0000010000000000) (0000001000000000) (0000000100000000) (0000000010000000) (0000000001000000) (0000000000100000) (0000000000010000) (0000000000001000) (0000000000000010) (0000000000000001) (0000000000000010) (0000000000000001)	Single bit errors (E_1)
(1100000000000000) (0110000000000000) (0011000000000000) (0001100000000000) (0000110000000000) (0000011000000000) (0000001100000000)	2-bit burst errors in the areas with higher and medium BER ($E_{B2}^{0..7}$)

Table 2. Vectors representing the errors to be detected in the considered example

Detectable errors (E_Δ)	Error subset
(1010000000000000) (1001000000000000) (0101000000000000)	2-bit random errors in the area with higher BER, excluding correctable errors ($E_2^{0..3} - E_{B2}^{0..3}$)
(1x10000000000000) (01x1000000000000)	3-bit burst errors in the area with higher BER ($E_{B3}^{0..3}$)
(0000000110000000) (0000000011000000) (0000000001100000) (0000000000110000) (0000000000011000) (0000000000001100) (0000000000001100) (0000000000000011) (0000000000000011)	2-bit burst errors in the areas with lower BER ($E_{B2}^{7..16}$)

Computing the Parity-check Matrix (H). After determining E_+ and E_Δ , the algorithm has to find an **H** matrix that satisfies the conditions (2) and (3) with the previously selected set of error vectors. As all the matrices have to be considered, it may require a huge computational effort. A recursive backtracking algorithm has been developed to lighten the process. It checks partial matrices and adds a new column only if the previous matrix satisfies the requirements.

In this way, the algorithm starts with a **partial_H** matrix, with $n-k$ rows and only one column. Then, it is checked that this matrix accomplishes the requirements. If it does, new columns are added recursively. Both the initial and the added columns must be non-zero, so there are $2^{n-k} - 1$ combinations for each column. The pseudo code of the partial matrix checker procedure is presented in Fig. 3.

```

Procedure CheckPartialMatrix partial_H  $(n-k) \times n_{cols}$  /*  $n_{cols} \in [1..n]$  */
  SyndromeSet = {}
  For each error vector e in  $E_+$ 
    partial_e =  $(e_1, e_2, \dots, e_{n_{cols}})$ 
    If HammingWeight(e) = HammingWeight(partial_e)
      newSyndrome = CalculateSyndrome(partial_H  $\times$  Transpose(partial_e))
      If newSyndrome in SyndromeSet then Return /* Not valid partial matrix */
      Else Add newSyndrome to SyndromeSet
    End if
  End for
  For each error vector e in  $E_\Delta$ 
    partial_e =  $(e_1, e_2, \dots, e_{n_{cols}})$ 
    If HammingWeight(e) = HammingWeight(partial_e)
      newSyndrome = CalculateSyndrome(partial_H  $\times$  Transpose(partial_e))
      If newSyndrome in SyndromeSet then Return /* Not valid partial matrix */
      Else Do Nothing /* newSyndrome not stored in this case */
    End if
  End for
  If  $n_{cols} = n$ 
    Add partial_H to SolutionsSet
    Return
  Else
    For each possible new_column /*  $n-k$  bits, excluding the all 0 combination:  $2^{n-k}-1$  possible values */
      CheckPartialMatrix [partial_H | new_column]  $(n-k) \times (n_{cols}+1)$ 
    End for
  End if
End procedure

```

Fig. 3. Partial matrix checker procedure

The Hamming weight indicates the number of 1s in a vector. It is used to check if the error vectors have all their 1s in the first n_{cols} bits, because it is impossible to calculate the syndrome for an error vector with 1s in columns not included in the partial matrix. As new columns are added, more error vectors can be processed.

The first loop generates syndromes for all selected error vectors, and they are added to **SyndromeSet**. When a new syndrome is generated, it is verified if it has been previously added. In this case, two different error vectors generate the same syndrome, so **partial_H** cannot be part of a valid solution. If the algorithm arrives at the end of the first loop, all selected error vectors have different syndromes.

Then, a second loop calculates syndromes for the detectable errors. Now, it is tested whether these syndromes have been previously added to **SyndromeSet** (i.e. the syndrome is associated to a correctable error). In this case, **partial_H** cannot be part of a valid solution. Unlike the first loop, the calculated syndromes are not added to **SyndromeSet**, as these syndromes are not associated only to one detectable error. If the algorithm arrives at the end of the second loop, all selected correctable errors have non-equal syndromes, and the detectable errors have distinct syndromes from those used for correction.

Now, if **partial_H** has n columns, then a solution has been found, and it is added to **SolutionsSet**. If it has fewer columns, a new column has to be added. A third loop for all possible combinations calls recursively the checker procedure.

Although the condition $|E_+| \leq 2^{n-k}$ was satisfied, it does not guarantee the existence of a code with the selected requirements. If no solution is found, **SolutionsSet** is empty, and hence the searched code does not exist. This only can be solved in two

ways: increasing the redundancy (i.e. a bigger value of n , maintaining k), or reducing the error control (that is, decreasing the number of error vectors to be corrected or detected). Once **SolutionsSet** is obtained, a solution can be chosen according to different criteria. Let us introduce here some of them:

- First found: this option allows reducing the **H** generation time.
- Smallest Hamming weight of **H**: this solution commonly offers circuits with the lowest number of logic gates in a hardware implementation, for example.
- Smallest Hamming weight of the heaviest row of **H**: the logic depth of each parity or syndrome bit generator circuit usually depends on the Hamming weight of the associated row. The heaviest row determines the speed of the encoder and the decoder circuits.

It is worth noting that other criteria can be applied, depending for example on the technology and requirements of the implementation of encoding and decoding.

Next paragraph presents the code obtained for our case study, applying the smallest Hamming weight of **H** criterion.

Code Implementation. Attending to the error vectors in Table 1 and Table 2, and selecting a solution with smallest Hamming weight in **H**, one possible solution is:

b/r_0	b/r_1	b/r_2	b/r_3	b/r_4	b/r_5	b/r_6	b/r_7	b/r_8	b/r_9	b/r_{10}	b/r_{11}	b/r_{12}	b/r_{13}	b/r_{14}	b/r_{15}	b/r_{16}
u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}	u_{11}	See (4)				
1	0	1	0	0	1	0	0	1	1	1	1	1	0	0	0	0
0	1	0	1	0	0	1	0	1	1	0	0	0	1	0	0	0
0	0	1	0	1	0	1	1	1	0	1	0	0	0	1	0	0
1	0	0	1	1	0	1	0	0	1	0	1	0	0	0	1	0
0	1	0	0	1	1	0	1	0	0	1	1	0	0	0	0	1

As stated in Section 2.1, once calculated **H**, encoding and decoding formulas can be obtained easily from it. In this case, **u** is part of **b**, and the parity bits are located in the columns with only one 1. Each parity bit is calculated by XORing the bits with a 1 in its row. For example, bit b_{13} of the codeword is a parity bit, because the corresponding column in **H** has only one 1, in the second row. Searching the other 1s in the same row, they are in the columns corresponding to u_1, u_3, u_6, u_8 and u_9 .

Similarly, **s** is calculated using **r**. Each row generates a syndrome bit by XORing the bits in positions with a 1 in each row. The formulas for the proposed code are:

$$\begin{aligned}
 & b_i = u_i, \forall i \in \mathbb{N} : 0 \leq i \leq 11 \\
 & b_{12} = u_0 \oplus u_2 \oplus u_5 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{11} & s_0 = r_0 \oplus r_2 \oplus r_5 \oplus r_8 \oplus r_9 \oplus r_{10} \oplus r_{11} \oplus r_{12} \\
 & b_{13} = u_1 \oplus u_3 \oplus u_6 \oplus u_8 \oplus u_9 & s_1 = r_1 \oplus r_3 \oplus r_6 \oplus r_8 \oplus r_9 \oplus r_{13} \\
 & b_{14} = u_2 \oplus u_4 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{10} & s_2 = r_2 \oplus r_4 \oplus r_6 \oplus r_7 \oplus r_8 \oplus r_{10} \oplus r_{14} \\
 & b_{15} = u_0 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_9 \oplus u_{11} & s_3 = r_0 \oplus r_3 \oplus r_4 \oplus r_6 \oplus r_9 \oplus r_{11} \oplus r_{15} \\
 & b_{16} = u_1 \oplus u_4 \oplus u_5 \oplus u_7 \oplus u_{10} \oplus u_{11} & s_4 = r_1 \oplus r_4 \oplus r_5 \oplus r_7 \oplus r_{10} \oplus r_{11} \oplus r_{16}
 \end{aligned} \tag{4}$$

The syndrome bits determine the next step. If they are all 0, **r** is assumed to be correct. If not, **s** can be associated to a correctable error. In this case, the bit(s) affected are corrected; otherwise, the “error detected” condition is achieved. Table 3 shows the association syndrome-correction/detection for the proposed example.

Table 3. Syndrome lookup table (estimated errors) for the proposed code

$s_4 s_3 s_2 s_1 s_0$	Error in...	$s_4 s_3 s_2 s_1 s_0$	Error in...	$s_4 s_3 s_2 s_1 s_0$	Error in...	$s_4 s_3 s_2 s_1 s_0$	Error in...
0 0 0 0 0	No error	0 1 0 0 0	bit r_{15}	1 0 0 0 0	bit r_{16}	1 1 0 0 0	Detection
0 0 0 0 1	bit r_{12}	0 1 0 0 1	bit r_0	1 0 0 0 1	bit r_5	1 1 0 0 1	bit r_{11}
0 0 0 1 0	bit r_{13}	0 1 0 1 0	bit r_3	1 0 0 1 0	bit r_1	1 1 0 1 0	bits r_6, r_7
0 0 0 1 1	Detection	0 1 0 1 1	bit r_9	1 0 0 1 1	Detection	1 1 0 1 1	bits r_0, r_1
0 0 1 0 0	bit r_{14}	0 1 1 0 0	Detection	1 0 1 0 0	bit r_7	1 1 1 0 0	bit r_4
0 0 1 0 1	bit r_2	0 1 1 0 1	bits r_4, r_5	1 0 1 0 1	bit r_{10}	1 1 1 0 1	Detection
0 0 1 1 0	Detection	0 1 1 1 0	bit r_6	1 0 1 1 0	bits r_3, r_4	1 1 1 1 0	Detection
0 0 1 1 1	bit r_8	0 1 1 1 1	bits r_2, r_3	1 0 1 1 1	bits r_1, r_2	1 1 1 1 1	bits r_5, r_6

These functions are easily implementable. Encoding is as simple as a XOR tree or equivalent circuitry or algorithm per parity bit. s is calculated in the same way. Correction can be implemented using a 5-to-32 binary decoder and some logic gates. It is important to note that no correction is performed in case of “error detection”.

Experimental Validation. The error coverage provided by this FUEC code has been evaluated by using a software-based implementation of its encoder and decoder circuits. Results show that all errors in E_+ are successfully corrected and all those in E_Δ are correctly detected. A similar experimentation has been carried out for all codes included in this paper.

4 Comparison with Existing Codes and Discussion

FUEC codes have some features which cannot be found in other codes, mainly the ability to deploy the desired error control functions in each part of a codeword. This hinders comparing them with other existing codes.

Let us consider the (72, 64) SEC-F7EC proposed in [1]. This UEC code has $E_+ = E_*^{0,6} \cup E_1^{7..71}$, i.e. it has full error correction in the strongly controlled area and single error correction in the weakly controlled area. Under different fault hypothesis, UEC alternatives are very limited, but FUEC codes allow a great flexibility.

Two FUEC codes proposals are presented here, with the same values of n and k (i.e. the same redundancy). Their encoder and decoder circuits induce similar area and temporal overhead, compared to the (72, 64) SEC-F7EC. The first one has also two areas, with $E_+ = E_0 \cup E_1 \cup E_{B2} \cup E_2^{0..6} \cup E_3^{0..6} \cup E_{B4}^{0..6}$, i.e. it corrects double and triple random errors, as well as 4-bit burst errors, in the strongly controlled area; and 2-bit burst errors in the weakly controlled area. Its parity-check matrix is:

$$H = \begin{bmatrix} 1100000 & 00010000000100101110110001101010101011001011001000000100100100010 \\ 1010000 & 01000000000100001001101100110101110100100001000111010101001001000 \\ 0101000 & 001001000001001000110010110101000000010101001101001000010010010100 \\ 0010100 & 00000001001000001001010101000011001000111000010010101010000100101 \\ 0001010 & 00000010010010100100001000100100100110000010010100011001010010110 \\ 0000101 & 001000101000010001000100100010000101000001011000010100000101001101 \\ 0000010 & 1000000010100001001010101000100010001001010100000100101011010110 \\ 0000001 & 00001000000010010000000001001010010010100101010110110110111011 \end{bmatrix}$$

The second FUEC alternative divides the codeword in three areas (of lengths 7, 20 and 45 bits), and with $E_+ = E_0 \cup E_1 \cup E_2^{0..6} \cup E_3^{0..6} \cup E_{B4}^{0..6} \cup E_{B5}^{0..6} \cup E_{B2}^{7..26} \cup E_{B3}^{7..26}$. The parity-check matrix obtained in this case is:

$$\mathbf{H} = \begin{bmatrix}
 1100000 & 01100100101101101100 & 100000000000110110100100010100010011010010010 \\
 1010000 & 10101100101011010010 & 01000000001000101010010000010000010100001000 \\
 0101000 & 00100001010100110100 & 001000001001101011001001001001001000001000100 \\
 0010100 & 00010101001100100110 & 000100000001011000111000111000100101100100000 \\
 0001010 & 00001000111001010111 & 000010000100111000000111100111100000011100000 \\
 0000101 & 00000010011010001101 & 00000100000000011111111100000011100000011101 \\
 0000010 & 10010010010010101001 & 000000100010000000000000011111111100000000011 \\
 0000001 & 01001001001001001100 & 00000001111000000000000000000000000001111111111
 \end{bmatrix}$$

These are just two examples of how single, multiple and burst error control can be combined in a code using our methodology, and how to divide the codeword in different number of areas, maintaining an accurate level of redundancy. In addition, encoder and decoder circuits are simple, fast and easy to implement from \mathbf{H} . These are the main features of our proposal.

On the other hand, the computational effort to calculate \mathbf{H} is a question being improved. The computational time required to completely execute the \mathbf{H} generation process, as all the possible $((n-k) \times n)$ \mathbf{H} matrices to check are $(2^{n-k}-1)^n$. In fact, as the backtracking algorithm generates partial matrices and uses branch & bound, a lot of combinations are discarded in early phases. However, a full algorithm execution might be unaffordable, especially with big values of n (≥ 64). Partial executions and some tricks can be used to reduce the execution time of the algorithm. Anyway, it only affects to the design time, but not to the encoding and decoding time.

5 Conclusions and Future Work

This paper presents Flexible Unequal Error Control (FUEC) codes. The purpose of these codes is to provide enough flexibility to establish any desired number of control areas in a codeword, and to deploy in each one the adequate error control capabilities, combining single, multiple and burst error correction and/or detection.

This challenge is of great interest when the bit error rate (BER) is variable along the same codeword. In addition to VHM, FUEC codes may also result of interest in automotive, aerospace or avionics industries, where different sources of interference, noise or process variations may result in areas with variable BER (vBER). In these applications, intermittent faults in VLSI circuits increase the fault rate in the affected bits. In addition, the occurrence of multiple faults makes necessary to consider diverse error patterns in data storage and transmission.

To show the capabilities of FUEC codes, some examples have been included in the paper. Also, we have shown the flexibility, feasibility and potentials of FUEC codes with regard to existing UEC codes. Moreover, the methodology proposed to generate FUEC codes allows optimizing the silicon area and/or the speed of the encoding and decoding circuits.

An important question related to FUEC codes generation is the computational time required to complete the algorithm. Although partial executions and sub-optimal solutions can be achieved, the performance of current version has to be improved. To cope with this challenge, a parallel version of the algorithm is currently under development. However, the FUEC codes included in the paper have been obtained using the sequential version.

A final important remark relates to the type of (spatial and/or temporal) variability exhibited by the BER in different scenarios. For instance, VHM presents only spatial variability, since the BER depends on the error position in the storage media. On the other hand, intermittent faults show both spatial and temporal BER variability, since errors may vary not only their location but also their frequency or duration. In this latter case FUEC codes are also necessary, but they require adaptive detection and tolerance mechanisms. This is indeed the next big challenge to cope with our research.

Acknowledgement. This work has been partially supported by the Universitat Politècnica de València under the Research Project DesTT (SP20120806), and the Spanish Government under the Research Project ARENES (TIN2012-38308-C02-01).

References

1. Fujiwara, E.: Code Design for Dependable Systems. John Wiley & Sons (2006)
2. Constantinescu, C.: Intermittent faults and effects on reliability of integrated circuits. In: Reliability and Maintainability Symposium (RAMS 2008), pp. 370–374 (January 2008)
3. Nightingale, E.B., Douceur, J.R., Orgovan, V.: Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In: Procs. ECCS (April 2011)
4. Chou, W., Neifeld, M.: Interleaving and error correction in volume holographic memory systems. *Applied Optics* 37(29), 6951–6968 (1998)
5. Namba, K.: Unequal error control codes with two-level burst and bit error correcting capabilities. *Electr. Comm. Japan III: Fund. Electronics Science* 87, 21–29 (2004)
6. Gil, P., et al.: Fault Representativeness. Deliverable ETIE2 of Dependability Benchmarking Project. IST-2000-25245 (2002)
7. Neubauer, A., Freudenberger, J., Kühn, V.: Coding Theory: Algorithms, Architectures and Applications. John Wiley & Sons (2007)
8. LaBel, K.A.: Proton single event effects (SEE) guideline, NEPP Program web site (August 2009), https://nepp.nasa.gov/files/18365/Proton_RHAGuide_NASA_Aug09.pdf
9. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. *IEEE Micro*. 23(4), 14–19 (2003)
10. Shamshiri, S., Cheng, K.T.: Error-Locality-Aware Linear Coding to Correct Multi-bit Upsets in SRAMs. In: IEEE International Test Conference, pp. 1–10 (November 2010)
11. Masnick, B., Wolf, J.: On linear unequal error protection codes. *IEEE Transactions on Information Theory* 13(4), 600–607 (1967)
12. Dutta, A., Toubia, N.A.: Reliable network-on-chip using a low cost unequal error protection code. In: Procs. DFT, pp. 3–11 (September 2007), doi:10.1109/DFT.2007.20

Safety Transformations: Sound and Complete?

Ute Schiffel

Reykjavík University
School of Computer Science
utes@ru.is

Abstract. Safety transformations transform *unsafe original software* into *safe software* that, in contrast to the unsafe version, detects if its execution was incorrect due to execution errors. Especially transformations based on arithmetic codes such as an AN- or ANB-code apply complex and error-prone transformations, while at the same time aiming for safety- or mission-critical applications. Testing and error injection are used so far to ensure correctness and error detection capabilities. But both are incomplete and might miss errors that change functionality or reduce error detection rates. Our research provides tools for a complete analysis of AN-encoding safety transformations. This paper presents our analysis tools and results for the AN-encoded operations. While we were able to demonstrate functional correctness, we discovered bugs that prevent propagation of errors almost completely for AN-encoded divisions and reduce propagation significantly for logical bitwise operations.

1 Introduction

The reliability of hardware is decreasing due to decreasing feature and increasing system size [5]. Thus, we need to be able to build reliable systems from unreliable (hardware) components.

Safety transformations transform *unsafe original software* into *safe software* that, in contrast to the unsafe version, detects if its execution was incorrect due to errors in the infrastructure used. This error detection facilitates error tolerance. Several safety transformations were developed recently:

- solutions using replicated execution and voting (e. g., [18,17])
- solutions based on arithmetic encoding (e. g., [21,19])

These solutions are about to enter the market for safety- and mission-critical systems, e. g., SIListra Systems (<http://www.silistra-systems.com>) is providing solutions based on replication and arithmetic encoding as well.

Objective. So far, safety transformations are just claimed and believed to be correct, since the applied techniques of replication or arithmetic encoding are well known. However, the implementation is complicated in many places. Hence, there is no guarantee that the resulting safe software produces the same results as the unsafe original software in an error-free execution or that the implementation ensures the best error detection theoretically possible. Thus, before applying these transformations in safety-critical applications, we need to check the following properties:

soundness of implementation i. e., original software and safe software are semantically equivalent as long as no error disturbs the execution.

completeness of error detection i. e., safe software indeed detects all execution errors that are determined by the technique implemented with the expected probability. A bad implementation of the error detection technique might prevent this. Note that safety transformations provide probabilistic error detection, i. e., they might miss errors with a certain probability.

To the best of our knowledge, existing research has assessed both problems by means of testing and error injections, e. g., [20] for arithmetic encoding or [7] for replication, arithmetic encoding and assertion-based techniques. However, testing as well as error injection are far from providing the completeness of verification required for safety- and mission-critical systems.

In contrast, formal methods promise completeness. Our goal is to evaluate how well formal methods are suited to verify soundness and completeness of safety transformations. However, safety transformations are still under active development. Thus, the verification techniques developed will have to be fully automated and suited for daily usage, e. g., in nightly tests, by developers that are not experts in formal verification. In this paper, we will focus on the AN-encoding as presented in [21] (see Section 2 for a short introduction). Due to the complexity of the task at hand we divided it into two parts:

1. Verify the safe implementations of basic AN-encoded operations.
2. Verify the actual safety transformation, i. e., replacing original instructions with AN-encoded versions of the instructions and generation of glue code.

This paper will focus on step 1 by verifying the soundness of implementation and partly the completeness of error detection of basic AN-encoded operations such as arithmetic operations (see Section 5). To make these verifications feasible we implemented an automated abstraction that reduces the number of test cases considerably (see Section 4). See the related work in Section 3 for our evaluation of the suitability of other methods for the described task.

Our contributions are

- Automated abstraction of the basic AN-encoded operations (arithmetic and logical operations, casts, bit modifications, comparisons, select instruction, and 128-bit integer arithmetic instructions) with the help of a user-supplied heuristic for abstracting integer constants used in the implementations.
- Complete testing of the soundness of implementation and error propagation capabilities of these abstracted AN-encoded operations' implementations.
- We found and diagnosed issues preventing error propagation for all divisions and all bitwise logical operations.

2 AN-Encoding in a Nutshell

Arithmetic codes add redundancy to processed data which results in a larger domain. This domain of possible words contains valid code words and non-code words. Arithmetic codes are conserved by correct arithmetic operations, i. e., a correctly executed operation taking valid code words as input produces valid

code words. On the other hand, faulty arithmetic operations destroy the code with a very high probability, i. e., result in a non-code word [3]. Operations executed on non-code words most likely produce non-code words.

The *AN-code* is one of the most widely known arithmetic codes. Encoding is done by multiplying data x_f stored in variable x with a constant A . Thereby, the encoded version $x_c = A * x_f$ is obtained. Only multiples of A are valid code words and every operation processing AN-encoded data has to conserve this property. Code checking is done by computing the modulus with A , which is zero for a valid code word. The code of any output of an AN-encoded program is checked.

For storing the AN-encoded version of the data, variable x is assigned a larger type than it had originally. An AN-encoded program processes exclusively AN-encoded variables. If a program originally adds two variables x and y with the values x_f and y_f , its AN-encoded version adds $x_c + y_c = A * x_f + A * y_f$ which is equal to $A * (x_f + y_f)$, which is a code word.

So, what is so complex about AN-encoding that we doubt the correctness of its implementation? Varying reasons make AN-encoded operations complex:

- ensuring behavior that is semantically equivalent to the unencoded version and produces an AN-code word, and
- operations not natively supported by AN-encoding.

AN-encoding was originally designed for arithmetic operations using infinite integers (no native support for floating point at all). However, the size of integers processed by CPUs is restricted and over- or underflows are happening. Most programs expect these over- and underflows to happen, e. g., the realization of signed integer operations with the two's complement relies on correct overflows. Ensuring this behavior for addition, subtraction and multiplication adds complexity. Furthermore, for some operations AN-encoding leads to intermediate results whose size is larger than the natively supported data types, so that software implementations for big, e. g. 128-bit, integers are required for AN-encoded multiplications and divisions.

Other operations such as logical operations or bit manipulations are not natively supported by AN-encoding and need to be implemented as AN-encoded versions that ensures error detection. Operations such as division are in principle supported by AN-encoding but additional instructions are required to ensure that the result is again an AN-code word and no intermediate decoding happens.

We are verifying AN-encoded operations that use 64-bit integers as encoded values and a 32-bit integer for the encoding parameter A . All operations exist for functional values of 8-, 16-, and 32-bit integers. The implementations for the different type sizes are equivalent apart from constants, e. g., constants used to correct the overflow behavior. See [24] for details on AN-encoded operations.

3 Related Work

Testing and Error Injection are used the most to verify the correctness of software. Testing exists in quite different flavors from unit to integration testing. A special form of testing for dependable systems is error injection which artificially introduces errors to an execution. However, even when using elaborate

test case generation tools, testing and error injection are always incomplete and can only be used for finding bugs but not for verifying the absence of bugs.

Static Analysis tools such as Polyspace by MathWorks [14]¹ and Astrée² aim at proving absence of runtime errors. They analyze source code to find bugs such as faulty pointer dereferences or infinite loops amongst others. However, they are not aiming for proving general correctness.

Formal Verification. In contrast to testing, formal verification, i. e., proving the correctness of software, is complete and establishes a high level of trust. However, it is time consuming and requires specialists. Even if powerful tools are used, these tools are usually interactive and require a specialist to guide the proof. However, we aim for a complete automation to be able to easily re-verify the software after each modification.

The verification of a C compiler can give us an idea, how expensive the verification of a safety transformation would be: Leroy presents the formally verified C compiler CompCert in [13] whose verification without verifying front- and back-end took five years according to the author.

However, some theorem provers can prove some problems completely automated if the problem is simple enough. Thus, they could be used to verify the correctness of AN-encoded operations. We tried to prove the simplest AN-encoded operation, the addition using KeY [1]. KeY works directly on Java code and, thus, would be easy to integrate in the development process. We would have preferred a tool for C, but we are not aware of one and translating the C-implementations of the AN-encoded operations into Java is a straight-forward process which requires only some string replacements. We are no theorem proving specialists and used the KeY tutorial [2] to provide the required verification conditions. However, KeY is not able to prove the addition without a request for help in picking a proof rule. We expect similar results for other theorem provers and other, more complex, AN-encoded operations.

Model Checking. Another verification technology promising completeness is model checking. Because the AN-encoded operations are implemented in C, we researched specifically model checkers that are able to check C code: LLBMC [15] (version 2012.2a) and CBMC [8] (version 4.2 from 2012). Both are bounded model checkers that transform the checked program and the negated property into a symbolic expression. For generating the symbolic expression, the checker unrolls loops and recursions n times. Thus, bounded model checking might be incomplete if a loop cannot be unrolled completely. However, the AN-encoded operations do contain neither loops nor recursions.

Then, the checker hands the generated symbolic expression to an SMT solver that checks if the expression is satisfiable. If so, a solution for the negated property was found, i. e., the property cannot always be true and, thus, was proven wrong. LLBMC uses the Boolector [6] SMT solver. CBMC supports several

¹ <http://www.mathworks.se/products/polyspace/>

² <http://www.astree.ens.fr/>

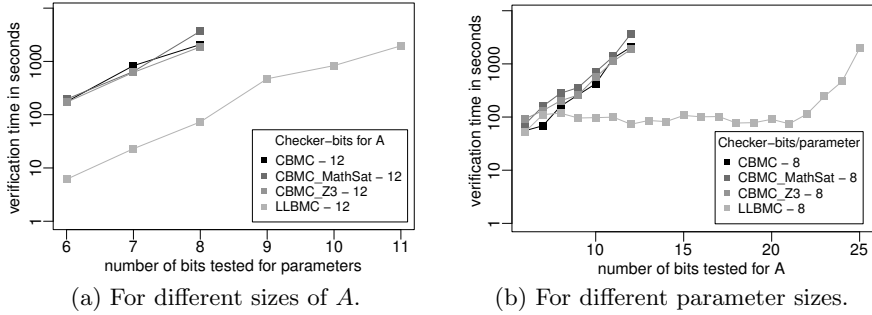


Fig. 1. Verification times of model checking the AN-encoded 32-bit addition with different model checkers and size restrictions for parameters and A

solvers: Z3 [9], MathSAT [10] and Boolector [6]. A similar but not available tool is Calysto [4] with its own SMT solver SPEAR.

We again tried to verify the AN-encoded addition, because it is the simplest of the AN-encoded operations. We used LLBMC and CBMC with different SMT solvers. However, symbolic expressions describing the correctness of AN-encoded operations contain non-linear arithmetic, which is not well supported by current SMT solvers. Thus, we expect problems due to state explosion. Indeed, for verifying the addition, we had to restrict the size of the three input parameters: two operands of the addition and the encoding parameter A . Thus, the verification is incomplete because many parameter combinations remained unchecked. We measured the runtime of these checks for different size restrictions. For both model checkers, the verification time is dominated by SMT solving. The results depicted in Figure 1 clearly show a state explosion with exponentially growing verification times (note that the y-axis is in log-scale) and the impossibility to solve this verification task completely using these checkers.

Similar results were obtained using Symbolic Execution in combination with the STP SMT solver as presented in [22]. Due to state explosion a complete verification was also not achieved.

Symbolic Error Injection. Several projects try to provide more complete results than simple error injection provides by using symbolic error injection: [11,16] They combine symbolic expressions of the analyzed software, error checks and possible errors into one model. Then, they check if all modeled errors are detected using model checking [16] or symbolic execution of the analyzed program [11]. However, both projects demonstrate their approaches only for small examples and quite restricted error models such as bit flips.

In [23] a system is presented which allows the user to model error injection experiments by describing error types and picking error injection points. The system will then automatically execute all possible injections. However, a complete test of reasonable sized systems is impossible.

4 Automatic Abstraction by Downscaling

As we described in Section 3, we were not able to verify the AN-encoded operations using formal or semi-formal verification approaches. The general idea of our alternative approach is to drastically reduce the number of test cases by automatically abstracting the AN-encoded operations and to execute complete tests using the resulting so-called *downscaled* operations. Our abstraction has to preserve the structure of code and processed data to ensure that bugs are not removed. See Figure 2 for an overview of our approach.

General Approach. Since the application domain of AN-encoded operations is restricted to transforming integer values using arithmetic operations and bit modifications, we can reduce the amount of test cases exponentially by using smaller integer types, e. g., instead of 64-bit integers, we can use 8-bit integers. Due to the structural equivalence of different integer types, downscaled implementations are functionally equivalent to unscaled implementations, that is, the downscaling transformation preserves all necessary properties of operations, such as overflows, underflows, special values, e. g. division by zero. However, the domain size and the under-/overflow bounds change when downscaling an m -bit integer type to an n -bit integer type:

- The overflow bound changes from 2^m to 2^n and
- The sign under-/overflow bound changes from 2^{m-1} to 2^{n-1} .

Implementation in LLVM. We implemented the necessary code transformations using the compiler framework LLVM [12]. We apply the following changes:

- downscale *types* of variables, function parameters and return values, and
- downscale values of *constants*.

We also change the type information of all LLVM instructions, which also results in the correct functional behavior according to the type, e. g., for overflows.

Furthermore, the pass supports, exclusion of data and functions calls from downscaling. We use this to support output operations such as `printf` in the downscaled code. All data depending on excluded data is also not downscaled. If a non-scaled function takes downscaled variables as input, these are up-casted to the original type for calls to this function.

Types. We support the downscaling of function types, integer types, pointer types and aggregate types such as arrays and structures. Since the AN-encoded

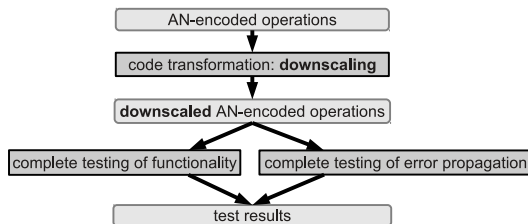


Fig. 2. System overview

operations do not use floating point types, we do not need to support their downscaling. Thus, we also do not need to downscale any instruction using floating point types. Note that currently AN-encoded programs that use floating points are realized using software-implemented floating point instructions.

For downscaling integer types, we reduce their size according to our used scale factor. For sizes below 8 bit, we use 8-bit integers and after all instructions executed on these values, we execute a modulo with $2^{typeSize}$ on the result to ensure correct overflow behavior.

For pointer types, we scale the pointed to type according to our rules. For aggregate types, we scale each element type separately. Thus, the number of elements of a structure does not change, only their size changes. For arrays, the number of elements is determined by a constant that is scaled according to the rules for scaling constants. Thus, the number of elements might change or not depending on the rules detailed in the next paragraph.

For function types, types of parameters and the type of the return value are scaled according to our rules and all calls are adapted except for functions that the user excluded from scaling.

Constants. How the values of constants are downscaled depends on their semantics, which only the user of our framework knows. Therefore, we provide a framework that helps the user of the downscaling transformation to

- *classify* the constant, and
- provide *rules* for its downscaling.

For classification, the user can use information about the constant, such as, its type, value, properties of the value (e. g. *is a power of two*) and context of usage (e. g. *used as a comparison for a jump*). He combines this information into a conditional expression that is associated with a transformation rule that is also provided by the user. The user builds these rules using basic rules that we provide, such as, *scale power of two* or *keep value but scale type*.

For downscaling all AN-encoded operations, we wrote 31 such classification-rule pairs. These facilitated the scaling of 604 constant usages. Note these might have the same value, but different contexts may result in different scaling.

5 Verifying Encoded Operations

We use the downscaled versions of the AN-encoded operations to execute complete tests for two properties: functional correctness and propagation of errors. The downscaling, generation of required test code and execution of tests is completely automated. Execution times are short enough to facilitate regular testing.

5.1 Complete Testing of Functional Correctness

Test Setup. For testing the functional correctness, we use the downscaled versions of AN-encoded operations. We test them completely, that is, we test them with all possible combinations of the downscaled encoding parameter A and the downscaled operation parameters that are valid code words for this A . Since the

encoded operations represent native operations of the processor, we use these native operations (for the matching downscaled type) to compute the expected result. We encode the expected result using the currently tested A and compare with the obtained result. If the values are not equivalent, we found a bug.

Result. For testing functional correctness, we scaled down all types by the factor 4. Testing and analyzing all 73 operations takes approximately 30 minutes on an Intel Core 2 Duo with 2.40GHz. We found no functional error.

5.2 Complete Testing of Error Propagation

Test Setup. Encoded operations should propagate errors that modified data to ensure the detection of these errors once the code of a result is checked. For testing error propagation, we test all possible inputs for parameters and A . We ignore test cases where all parameters are valid code words for the tested A , because we are interested in *how well errors propagate to the output*. Using the tested A , we check if the obtained result is a valid code word. If so, a non-propagation was detected. Note that a certain amount of not propagating errors is normal, because an AN-code detects errors only with a probability of approximately $1 - \frac{1}{A}$. Thus, we have to check if the amount of not propagated errors is considerably larger than expected.

For complete error propagation testing, we scaled down by factor 8, because factor 4 did not reduce the number of test cases sufficiently. This resulted in non-native types such as 4-bit integers. After operations on these, we add explicit modulo operations to ensure the correct overflow behavior.

Result. The question is: How do propagation rates for downscaled and unscaled implementations relate to each other? Can we draw conclusions from these measurements? To estimate the differences, we executed randomized propagation tests for additions and divisions for versions downscaled by 2 and 4, and the original non-scaled implementations. For each scale factor, we executed 400000 tests for 100 different A s for both operations. Figure 3 shows exemplary results. We observe:

1. Measurements for downscaled implementations seem to provide a good prediction for the propagation rate of non-scaled implementations, since all curves are similar.
2. Furthermore, while for the addition the propagation probability relates as expected to A , the division does not propagate most errors.

We executed complete propagation tests for all operations. For most operations, we see non-propagation rates similar to the approximately expected $\frac{1}{A}$. However, signed and unsigned divisions for all types do not propagate most of the errors. Furthermore, the 16-bit bitwise logical operations starting with a certain size of A are propagating less errors than theoretically expected. See Figure 4 for the results for some of the operations.

We identified the reason for the reduced propagation for the logical operations by looking at their source code: For understanding, it is necessary to know that, due to performance reasons, the logical operations are not executed AN-encoded

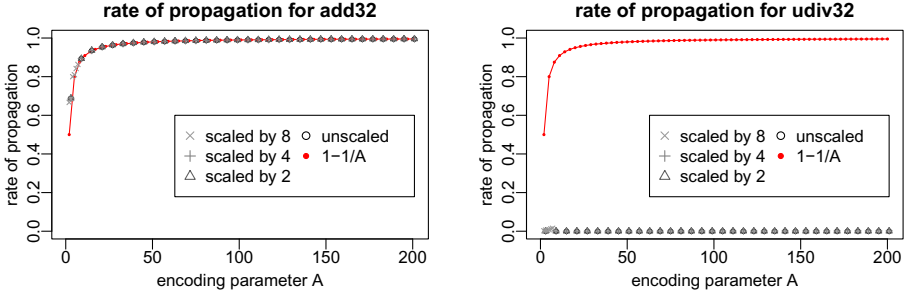
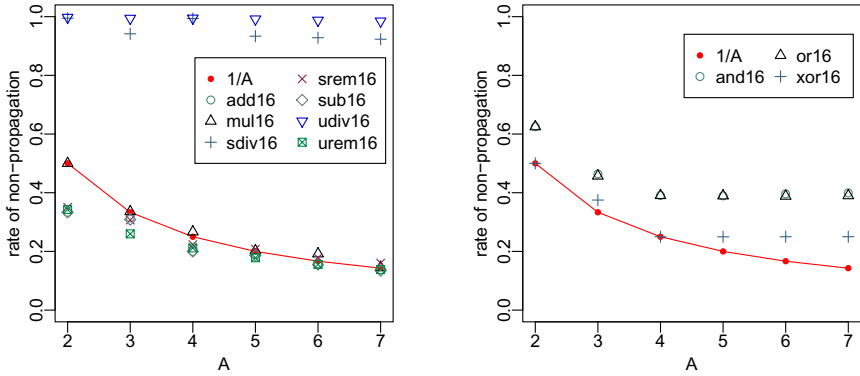


Fig. 3. Propagation rates for unscaled implementations and implementations scaled by factors 2, 4 and 8, and approximately expected propagation rate $1 - \frac{1}{A}$



(a) Arithmetic operations.

(b) Logical operations.

Fig. 4. Rate of not propagating errors (for a selection of operations)

but replicated within one variable of sufficient size. This requires transforming the AN-encoded operands into this format and transforming the result back into the AN-code. This is done in a seemingly safe way, that is, for a non-code word we would expect unequal replicas. However, the larger A is the more probable it is that during this transformation overflows from one replica into the other neutralize erroneous input. In addition, in contrast to `xor`, `and` and `or` neutralize some mismatches in the replicas. We clearly see this effect in Figure 4b: `and` and `or` have higher rates of not propagating errors than `xor`.

To aid debugging non-propagation, we plotted the parameter combinations for which no error propagation occurred. For exemplary results, see Figure 5. For easier accessibility, we reduced the graphs here to the domain of values that represent functional values (identified by the rectangular markings) and left out code words that represent larger than permitted functional values. Note that for

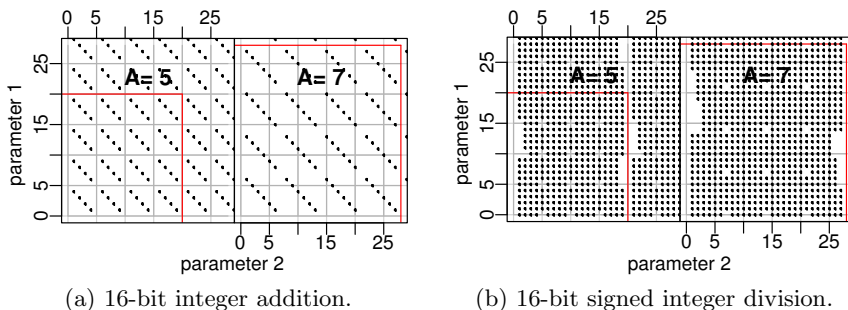


Fig. 5. Invalid inputs that result in outputs that are code-words, exemplary for addition and signed division. Note that the domains are also downscaled.

the propagation rate generally also the non-depicted parts of the input domains matter since an error could easily result in these values.

For the **addition**, we see a pattern that tells us that non-propagation is the result of the two parameters adding up to a multiple of A , i. e., an AN-code-word. As expected, with increasing A the space between not propagated combinations grows. However, for the **division**, we see that most of the combinations do not propagate. There are only a few combinations where propagation occurs. Careful analysis of the patterns helped us to identify and fix the problem which was caused by the code that ensured the divisibility of the AN-code words.

Executing and analyzing (i. e., generating the plots) the full propagation tests for all 16- and 32-bit operations takes 27 minutes. If we leave out the three-operand select operation, we can reduce the time to 14 minutes.

6 Conclusion

Summary and Results. We presented our fully automated simple abstraction which facilitates complete testing. Furthermore, we compared our approach to formal and semi-formal approaches. Note that the simplicity of the approach comes at the price: the generalization is limited. However, the framework can be used to test other complex libraries with a clearly defined interface, for example, the SafeInt library, or the library of ANB-encoded operations, or software-implemented floating point operations.

Applying our automatic abstraction to AN-encoded operations used by the safety transformation described in [21], we completely verified the functional correctness of these implementations. In contrast to existing formal and semi-formal approaches, we were also able to use the same approach to completely test the error propagation properties of these implementations. Thereby, we found and diagnosed two serious flaws that reduced error propagation, in one case even prevented propagation completely. We diagnosed both bugs with the help of our graphical representation of the results.

Future Work. Our next steps will be to build on these results and use and adapt translation validation approaches to prove the complete AN-encoding safety transformation correct (or find flaws in it).

Acknowledgements. This research was supported by START grant 120056041 by *The Icelandic Centre for Research*.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. *Software and System Modeling* 4, 32–54 (2005)
2. Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., Schmitt, P.H.: Verifying Object-Oriented Programs with KeY: A Tutorial. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) *FMCO 2006*. LNCS, vol. 4709, pp. 70–101. Springer, Heidelberg (2007)
3. Avizienis, A.: Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. In: *IEEE Transactions on Computers*, pp. 1322–1331 (1971)
4. Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: *Proceedings of the 30th International Conference on Software Engineering, ICSE 2008*, pp. 211–220. ACM, New York (2008)
5. Borkar, S.: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*. 25(6), 10–16 (2005)
6. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Kowalewski, S., Philippou, A. (eds.) *TACAS 2009*. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
7. Chang, J., Reis, G.A., August, D.I.: Automatic Instruction-Level Software-Only Recovery. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 83–92. IEEE Computer Society, Washington, DC (2006)
8. Clarke, E.M., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 8, 1–27 (2012)
11. Larsson, D., Hähnle, R.: Symbolic Fault Injection. In: *4th International Verification Workshop in connection with CADE-21, Bremen, Germany*, pp. 85–103 (2007)
12. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, p. 75. IEEE Computer Society, Washington, DC (2004)
13. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* 52, 107–115 (2009)
14. The MathWorks. Code Verification and Run-Time Error Detection Through Abstract Interpretation. Technical report, The MathWorks (2012)

15. Merz, F., Falke, S., Sinz, C.: LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 146–161. Springer, Heidelberg (2012)
16. Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: SymPLFIED: Symbolic program-level fault injection and error detection framework. In: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, DSN 2008, pp. 472–481 (June 2008)
17. Reis, G.A., Chang, J., August, D.I., Cohn, R., Mukherjee, S.S.: Configurable Transient Fault Detection via Dynamic Binary Translation. In: Proceedings of the 2nd Workshop on Architectural Reliability (2006)
18. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 243–254. IEEE Computer Society, Washington, DC (2005)
19. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and aNBDmem-encoding: Detecting hardware errors in software. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 169–182. Springer, Heidelberg (2010)
20. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: Software-Implemented Hardware Error Detection: Costs and Gains. In: The Third International Conference on Dependability, DEPEND 2010 (2010)
21. Fetzer, C., Schiffel, U., Süßkraut, M.: AN-encoding compiler: Building safety-critical systems with commodity hardware. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 283–296. Springer, Heidelberg (2009)
22. Seeleemann, M.: Evaluation of Predicate-Complete Test Coverage and Symbolic Execution for Software Testing and Verification. Master's thesis, Technische Universität Dresden (2010)
23. Svenningsson, R., Eriksson, H., Vinter, J., Törngren, M.: Generic fault modelling for fault injection. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 287–296. Springer, Heidelberg (2011)
24. Wappler, U., Fetzer, C.: Hardware Failure Virtualization Via Software Encoded Processing. In: 5th IEEE International Conference on Industrial Informatics, INDIN 2007 (2007)

It Is (Almost) All about Human Safety: A Novel Paradigm for Robot Design, Control, and Planning

Sami Haddadin, Sven Parusel, Rico Belder, and Alin Albu-Schäffer

Robotics and Mechatronics Center, DLR

Abstract. In this paper we review our work on safe control, acting, and planning in human environments. In order for a robot to be able to safely interact with its environment it is necessary to be able to react to unforeseen events in real-time on basically all levels of abstraction. Having this goal in mind, our contributions reach from fundamental understanding of human injury due to robot-human collisions as the underlying metric for “safe” behavior, various interaction control schemes that ground on the basic components impedance control and collision behavior, to safe real-time motion planning and behavior based control as an interface level for task planning. Based on this foundation, we also developed joint interaction planners for role allocation in human-robot collaborative assembly, as well as reactive safety oriented replanning algorithms. A very recent step was the development of novel programming paradigms that act as a simple yet powerful interface between programmer, automatic planning, and the robot. A significant amount of our work on robot safety and control has found its way into international standardization committees, products, and was applied in numerous real-world applications.

1 Introduction

Finally, first robotic systems gained sufficient control capabilities to perform delicate and complex manipulation and physical human-robot interaction (pHRI) tasks that require the dynamic exchange of physical forces between the robot and its environment. The fully torque-controlled DLR Lightweight Robot III (LWR-III) is such a device [1] and was recently commercialized by the robot manufacturer KUKA (KUKA LWR) [4]. This step made it possible to automate difficult and up to now still manually executed assembly tasks. In particular, the achieved sensitive and fast manipulation capabilities [3,14,20,23] of the robot prevent damage from the handled potentially fragile objects and humans directly interacting with the device. Recently, there is strong interest in making classical safety barriers, as e.g. fences or light barriers, obsolete for these interactive devices in order to enable direct physical cooperation between human and robot. For understanding the risks of this undertaking we performed a series of safety investigations [10,9,11,8,12,13,21], which led to fundamental insight into the potential injury a human would suffer due to a collision with a robot. Furthermore, we developed human-friendly interaction control and motion schemes

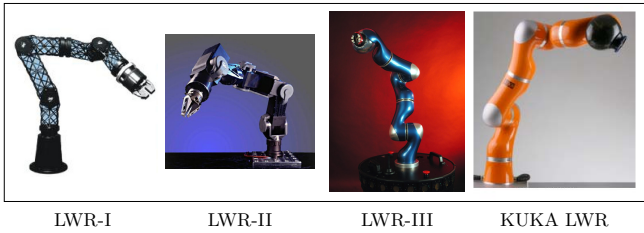


Fig. 1. The generations of DLR light-weight robots (LWR-I, LWR-II, and LWR-III) and the commercialized version (KUKA LWR)

that enable the robot to show sophisticated real-time responses on interaction force level, motion planning, and real-time task planning [24,6,14,18,15,22,16]. Generally, our approach of embodying reactivity on all levels of robot design and control is to our understanding the core to safe acting and manipulation in human environments. Consequently, the careful design and selection of methods that satisfy this requirement was our main premise.

In this paper we give an overview of the developed analysis tools, control schemes, motion planners, real-time behaviors, interaction planning, and programming paradigms for robots that are sought to act and manipulate in human environments. We intend to give a “bird’s eye” view on the available repertoire of tools and how the developed methodologies, insights, and algorithms impact robotics in general.

2 Technologies and Methods

2.1 Lightweight and Mechatronic Robot Design

The most basic step for building robots that interact with dynamic environments is to design them compact, light-weight, and with high payload. Only light structures are capable of appropriate physical reaction to external forces, i.e. have low intrinsic impedance. Secondly, the robot’s proprioceptive sensorization is a key element. Apart from standard motor position sensing, joint torque sensing together with accurate flexible joint dynamics modeling enable real torque control and the sensation of contact forces. In this line of thinking we have developed a series of torque controlled lightweight robots at DLR that are suitable for a diverse range of applications involving space, industry, medical, and domestic use. Figure 1 shows the history of the DLR Lightweight robots, resulting in its commercialized version: the KUKA LWR [4] (and more recently the LBR iiwa). Apart from minor modifications, this manipulator has exactly the same design as the 3rd generation of the DLR Lightweight robots [1], which are kinematically redundant, 7-DoF, joint-torque controlled flexible joint robots. The current version is the result of 15 years of research that produced three consecutive generations. Since the LWR-III weighs 13.5 kg and is able to handle loads up to

15 kg, an approximate load-to-weight ratio of 1 is achieved¹. The robot is a modular system and the joints are linked via carbon-fiber structures. The electronic parts, including power converting elements are integrated into the structure of the arm. Each joint is equipped with a motor position and a joint-torque sensor. Additionally, a 6-DoF force sensor can be embedded in the wrist. All electronics, motors, and gears are integrated into the arm, which makes the robot very compact and portable.

2.2 Interaction and Manipulation Control

Apart from reducing the reflected mechanical impedance of a robot in order to “make the mechanics sensitive”, the design of interaction control schemes is an essential step for sensitive force exchange with the environment. The most widely used control approach to physically interact with robots is probably impedance control and its related schemes, introduced in the pioneering work of Neville Hogan [19] and extended to flexible joint robots in [7,2,26,3,20]. This type of controller imposes a desired physical behavior with respect to external forces on the robot. For instance the robot is controlled to behave like a Cartesian second order mass-spring-damper system, see Fig. 2.

$$\mathcal{F}_{\text{ext}} = M_x(\ddot{\mathbf{x}} - \ddot{\mathbf{x}}_d) + D_x(\dot{\mathbf{x}} - \dot{\mathbf{x}}_d) + K_x(\mathbf{x} - \mathbf{x}_d), \quad (1)$$

where $\mathbf{x}, \mathbf{x}_d \in \mathbb{R}^6$ are the current robot and desired tip position, $\mathcal{F}_{\text{ext}} \in \mathbb{R}^6$ is the external wrench and $M_x, K_x, D_x \in \mathbb{R}^{6 \times 6}$ are the desired Cartesian inertia, stiffness, and damping tensors². Consequently, impedance control allows to realize compliance of the robot by means of active control. Interaction with an impedance controlled robot is robust and intuitive, since in addition to the commanded trajectory, a (local) disturbance response is defined. A major advantage of impedance control is that discontinuities like contact-non-contact do not create such stability problems as they occur with for example hybrid force control [5]. However, important open questions still need to be tackled from a control point of view, such as how to automatically and/or adaptively adjust the impedance parameters according to the current task. First work in this direction can be found in [25,23].

Apart from nominal interaction control, a robot sharing its workspace with humans and physically interact with its environment should be able to quickly detect collisions and safely react to them. In the absence of external sensing, relative motions between robot and environment/human are unpredictable and unexpected collisions may occur at any location along the robot arm. Various algorithms for coping with this problem were developed and evaluated. Efficient

¹ Please note that the nominal payload for the KUKA LWR is 7 kg, but it is able to handle up to 15 kg for research purposes.

² Please note that for the LWR-III we leave the inertia unshaped in order to preserve passivity of the controller. In turn, damping design becomes an important issue since the eigenfrequency is due to the Operational space mass matrix position dependent. Details can be found in [3].

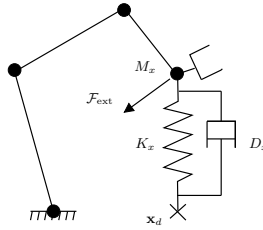


Fig. 2. Desired mechanical behavior expressed by mass-spring-damper

collision detection methods that use only proprioceptive robot sensors and provide also directional information for a safe robot reaction after collisions were introduced and validated [6,14].

Since our collision detection method gives not only binary contact information but an accurate estimation of the external torques, this information can be used to classify the sensed contact according to its severity. Based on this information it is possible to design application specific reaction patterns that are automatically executed if the required stimulus is sensed. Basically, a severity mapping $sm : \tau_{\text{ext}} \rightarrow s$ can be designed either as a fixed *stimulus type* \rightarrow *reaction* or a rather complex decision algorithm. In particular, this local interpretation of contact can classify the intensity and hardness of the contact based on contact frequencies and force amplitudes. This enables the robot to act locally very quickly, if unexpected interaction forces occur and act according to specified patterns (some details on this are given in Sec. 2.6). This can e.g. be used for activating automatic recovery strategies during identified failed grasping of objects, especially for avoiding the risk of damaging them.

The Cartesian impedance controller as well as the collision detection and reaction methods are already integrated in the KUKA LWR, i.e. available as a commercial product. Important to notice is that these novel features are considered as the key to enable safe pHRI by industry.

2.3 Injury Based Safety Analysis

During unexpected collisions with humans, various injuries, e.g. due to fast blunt impacts, dynamic and quasi-static clamping, or cuts by sharp tools may occur. In order to assemble a larger picture of this problem, we discussed and analyzed various worst-case scenarios in pHRI according to the following scheme

1. Select and/or define and classify the impact type
2. Select the appropriate injury measure(s)
3. Evaluate the potential injury of the human
4. Quantify the influence of the relevant robot parameters
5. Evaluate the effectiveness of countermeasures for injury reduction and prevention



Fig. 3. Collision experiments with an LWR-III, HIII dummy (upper) and human (lower)

Attempts to investigate real-world threats via impact tests at standardized crash-test facilities and to use the outcome to analyze safety issues during physical human-robot interaction were carried out. In order to quantify the potential danger emanating from the LWR-III, impact tests at the Crash-Test Center of the German Automobile Club (ADAC) were conducted and evaluated, see Fig. 3 (upper). Consecutive work extended the initial analysis for various other robot types, clamping, and even to sharp contact [16], see Fig. 3 (lower). Generally, the analysis provides unique data that helps explaining the characteristics of robot-human impacts, which in turn can be used for safer robot design and control as described next. Furthermore, the results are used as an input for future service robotics standards that define “safe” behavior of robots in human environments.

2.4 Biomechanically Safe Velocity

As already explained, the definition of injury, as well as understanding its general dynamics are essential in order to quantify what safe behavior really means. This insights can then be applied to control robots such that injury prevention is explicitly taken into account. For systematically bridging this gap, we approached the problem from a medical injury analysis point of view in order to formulate the relation between robot mass, velocity, impact geometry, and resulting injury qualified in medical terms [16]. We transformed these insights into processable representations and propose a motion supervisor that utilizes injury knowledge for generating safe robot motions. The algorithm, coined Safe Motion Unit (SMU), takes into account the reflected inertia, velocity, and geometry at possible impact locations. The proposed framework forms a basis for generating

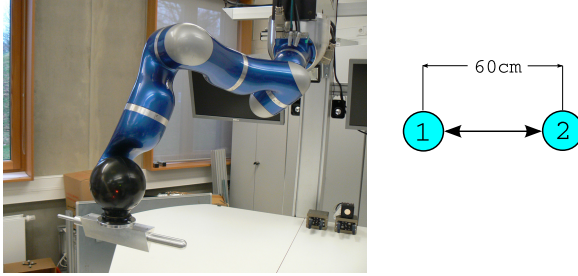


Fig. 4. DLR Lightweight-Robot III equipped with the end-effector that is used in the experiments (left). Trajectory for the “line test” (right).

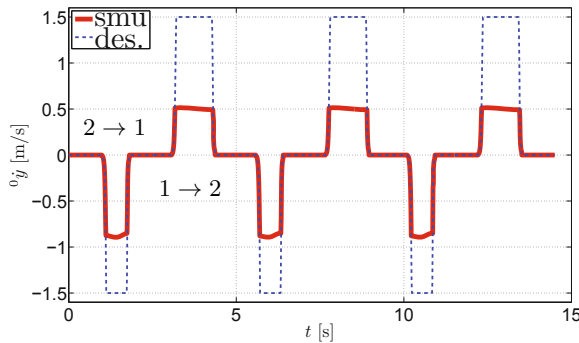


Fig. 5. SMU experiment “line test” with a Cartesian point-to-point motion

truly safe velocity bounds that explicitly consider the dynamic properties of the manipulator and human injury.

Figure 4 and Fig. 5 give an example for such a velocity scaling, where an LWR equipped with a possibly dangerous endeffector is commanded to move on a straight line between two configurations. The desired velocity is set to 1.5 m/s, whereas the SMU scales down the velocity such that according to its internal injury knowledge no injury would occur if the robot would accidentally collide with a human. The basic idea of our method currently finds its way into an ISO technical specification that defines safety requirements for collaborative industrial robots.

2.5 Real-Time Motion Planning

Up to now, we discussed rather the design and low-level control schemes for our robots. However, the real-time planning and execution of motions in a dynamic and partially unknown environment is fundamental for autonomous and safe



Fig. 6. Automatic recovery from physical collisions with real-time motion planning



Fig. 7. Real-time motion planning at 500 Hz for a global 3-goals motion planning problem

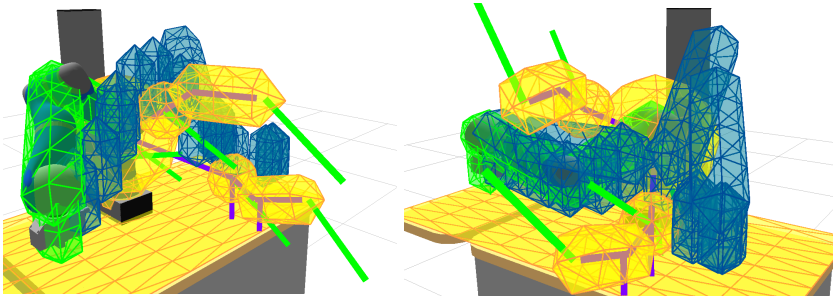


Fig. 8. Real-time motion planning and full-arm collision avoidance for dynamically moving obstacles

acting. If contact is desired or inevitable, also motion planning should be able to robustly and safely handle it, see Fig. 6. However, typically this is only approached as a pure control problem. Nonetheless, we believe this to be a rather artificial separation that misses the chance of designing more sophisticated responses to contact on trajectory level as well. Especially physical Human-Robot Interaction (pHRI) is a field in which such behavior is certainly desired. As human and robot shall collaborate very closely, the problem of generating “human-friendly” motions is of large interest. We developed several methods for dealing with obstacles and contact in real-time [18,15] on motion planning level. We could show for several problems, which were typically a domain for global sampling based planners that they can be solved in hard real-time³ with local methods only, see Fig. 7. This is due to the fact that these algorithms have favorable convergence properties. Another key feature of these schemes is the unified treatment of virtual and physical forces, which allows the systematic fusion of obstacle avoidance with collision retraction or exploratory tactile behavior, see Fig. 6.

Fig. 8 shows our more recent results on extending the schemes with predictive multi-agent systems that evaluate candidate paths in real-time and produce significantly better results (right) compared to the original version of the algorithm (left). In particular, a set of basic task related cost functions facilitate the separation of good candidate paths from less favorable ones.

2.6 Behavior Based Control for Safe Acting and Manipulation

Due to the diversity and complexity of the developed control capabilities and their sheer number it is non-trivial to design, implement and switch between them consistently under the premise of ensuring safe behavior. For that reason we developed a control architecture and a formal representation structure for interactive robots, which contains and consistently combines a wide set of

³ Our current implementation runs at 500 Hz. Presumably, the high parallelizability of our algorithm will enable us to further speed up the scheme.

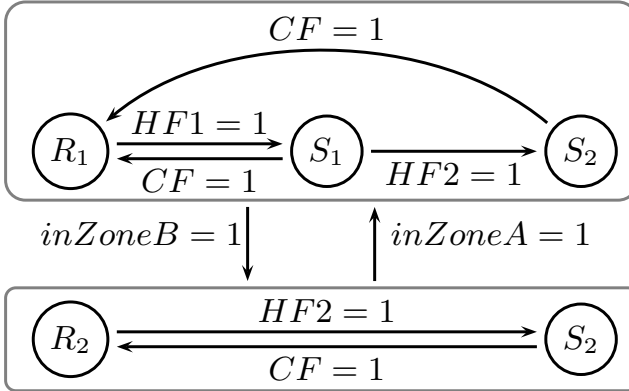


Fig. 9. Simple discrete planner for realizing context sensitive behavior of a robot. This example scheme enables the robot to behave differently during free motion and object manipulation phase. In this example R_1, R_2 are the nominal behaviors in *zone A* and *zone B*, respectively. $inZoneA$ and $inZoneB$ indicate whether the robot is operating in free space or close to the object, defined by a encapsulated surface of certain maximum distance to the object. S_1 denotes the safety reflex behavior for stopping abruptly and S_2 for switching to torque control with gravity compensation. CF , $HF1$, and $HF2$ denote *human confirmation* and *contact severity level*.

strategies for safe manipulation and human-friendly behavior [17,22]. We designed an encapsulated low-level control framework, which provides a discrete atomic action interface, which smallest primitive is defined as *atomic action* := (*command, behavior*). *command* can be e.g. *atomic-move2*, *switch-behavior*, or a simple *stop*. This is a rather classical approach. However, in contrast to other robots, the *behavior* is in our case a very complex data structure that defines the “overall” control activation the robot occupies. It defines a minimal representation of the activated interaction, motion, and local decision capabilities of the robot. This intuitive level of abstraction gives the task programmer or task planner a very powerful interface to the robot. Furthermore, we distinguish between *operational behavior* and *reflex behavior*.

- **Operational behaviors:** a formal high-level parametrization of the robot capabilities that defines its particular motion, control, and safety properties. This fully determines the nominal motion control and disturbance response of a robot. The atomic components of any general task automaton are *operational behaviors*.
- **Reflex behaviors:** a formal parameterization of a real-time reflex behavior of a robot that is associated with a real-time activation signal. This represents either the indication of a certain stimulus or a fault⁴. Reflexes override

⁴ Stimuli are general perception inputs, whereas faults are detected either by processed stimuli (observation of external torques, proximity information, ...) or general system malfunctions, as e.g. communication collapse or run-time violations.

the currently active operational behavior and execute a low-level strategy. Complex reflex patterns are directed reflex graphs, which represent a decisional component in the inner most control loop.

Figure 9 depicts a simple example for illustrating the concept. Generally, the described approach intends to tightly couple the *block world* and *control world*, i.e. leaving the common separation based designs. The presented design is from our point of view a missing link between control and task planning for interactive robots.

2.7 Joint Assembly and Interaction Planning

In order to profit from the collaboration of human and robot by combining the flexibility, knowledge and sensory skills of a human with the efficiency, strength, endurance and accuracy of a robot, according interactive assembly planners were developed to plan their joint actions for a common goal. For this, a basic question to be addressed is how high-level actions need to be assigned to human and robot, respectively. We developed a formal problem formulation for human-robot task allocation in the context of assembly tasks and analyzed standard optimization techniques from state-space search with respect to their applicability and performance characteristics. These methods found also their way into our integrated robot control architecture, see Fig. 10.

Apart from planning joint plans, it is crucial to equip robots with capabilities to perform local task replanning quickly and safely, as various faults may arise in the course of action. Exploiting, however, the complex capabilities of sophisticated interaction control schemes also on a decisional level was treated only marginally so far. Figure 11 depicts our approach to the problem of

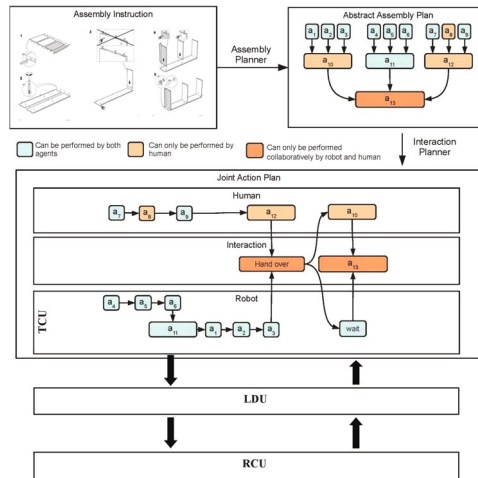


Fig. 10. Joint Assembly and Interaction Planning

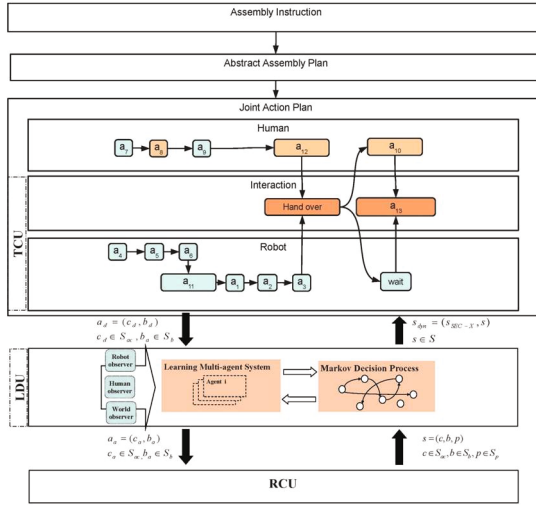


Fig. 11. Reactive Robot Control Framework

dynamic action and behavior learning, adaptation, and selection. We developed an algorithmic framework for learning high-dimensional, interactive robot actions based on an extended version of optimal adaptive learning for extensive support of dynamic, however, still human-friendly action generation. The scheme utilizes a concept for modeling interaction based on an interaction world and safety related metrics (similar to the ones for safe velocity planning). In addition, we designed an online behavior selection and adaptation algorithm that enables the robot to locally adapt its behavior such that human safety can be ensured in case of undesired and potentially dangerous events. The developed framework intends to bridge the gap between non-realtime task/interaction planners and hard real-time robot control algorithms for complex robotic systems.

3 Dynamic Programming Paradigms

Since planning of complex tasks is still at an early stage, robot programming on task level is still a major topic, in particular for interactive robots. In order to program tasks involving manipulation and interaction for such complex robots as the LWR, it is also important to be able to easily integrate new planning and perception components. Furthermore, robot programming needs to be intuitive, but yet powerful and flexible. The simple programming of reactive action generation patterns and their encapsulation is a highly desirable feature for reuse of already designed control programs. We developed a robot programming software framework that allows for designing control programs and distributed computation on various levels of abstractions and, if desired, with various underlying paradigms. see Fig. 12. It supports parallelism, seamless hierarchy, flexible

Acknowledgment. This work has been partially funded by the European Commission's Seventh Framework Programme as part of the project SAPHARI under grant no. 287513.

References

1. Albu-Schäffer, A., Haddadin, S., Ott, C., Stemmer, A., Wimböck, T., Hirzinger, G.: The DLR lightweight robot - lightweight design and soft robotics control concepts for robots in human environments. *Industrial Robot Journal* 34(5), 376–385 (2007)
2. Albu-Schäffer, A., Ott, C., Hirzinger, G.: A passivity based cartesian impedance controller for flexible joint robots - Part II: Full state feedback, impedance design and experiments. In: *Int. Conf. on Robotics and Automation (ICRA 2004)*, New Orleans, USA, pp. 2666–2673 (2004)
3. Albu-Schäffer, A., Ott, C., Hirzinger, G.: A unified passivity-based control framework for position, torque and impedance control of flexible joint robots. *The Int. J. of Robotics Research* 26, 23–39 (2007)
4. Bischoff, R., Kurth, J., Schreiber, G., Koeppe, R., Albu-Schäffer, A., Beyer, A., Eiberger, O., Haddadin, S., Stemmer, A., Grunwald, G., Hirzinger, G.: The kuka-dlr lightweight robot arm: a new reference platform for robotics research and manufacturing. In: *International Symposium on Robotics (ISR 2010)*, Munich, Germany (2010)
5. Craig, J., Raibert, M.: A systematic method for hybrid position/force control of a manipulator. In: *IEEE Computer Software Applications Conf.*, pp. 446–451 (1979)
6. De Luca, A., Albu-Schäffer, A., Haddadin, S., Hirzinger, G.: Collision detection and safe reaction with the DLR-III lightweight manipulator arm. In: *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2006)*, Beijing, China, pp. 1623–1630 (2006)
7. Goldsmith, P.B., Francis, B., Goldenberg, A.: Stability of hybrid position/force control applied to manipulators with flexible joints. *Int. Journal of Robotics and Automation* (14), 146–159 (1999)
8. Haddadin, S., Albu-Schäffer, A., Frommberger, M., Rossmann, J., Hirzinger, G.: The “DLR Crash Report”: Towards a standard crash-testing protocol for robot safety - part I: Results. In: *IEEE Int. Conf. on Robotics and Automation (ICRA 2008)*, Kobe, Japan, pp. 272–279 (2009)
9. Haddadin, S., Albu-Schäffer, A., Hirzinger, G.: Dummy crash-tests for the evaluation of rigid human-robot impacts. In: *IARP International Workshop on Technical Challenges and for Dependable Robots in Human Environments (IARP 2007)*, Rome, Italy (2007)
10. Haddadin, S., Albu-Schäffer, A., Hirzinger, G.: Safety evaluation of physical human-robot interaction via crash-testing. In: *Robotics: Science and Systems Conference (RSS 2007)*, Atlanta, USA, pp. 217–224 (2007)
11. Haddadin, S., Albu-Schäffer, A., Hirzinger, G.: Requirements for safe robots: Measurements, analysis & new insights. *The Int. J. of Robotics Research* 28(11-12), 1507–1527 (2009)
12. Haddadin, S., Albu-Schäffer, A., Hirzinger, G.: Soft-tissue injury in robotics. In: *IEEE Int. Conf. on Robotics and Automation (ICRA 2010)*, Anchorage, Alaska, pp. 3462–3433 (2010)
13. Haddadin, S., Albu-Schäffer, A., Hirzinger, G.: Soft-tissue injury caused by sharp tools: Definitions, experiments and countermeasures. Accepted at: *IEEE Robotics and Automation Mag.* (2011)

14. Haddadin, S., Albu-Schäffer, A., Luca, A.D., Hirzinger, G.: Collision detection & reaction: A contribution to safe physical human-robot interaction. In: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2008), Nice, France, pp. 3356–3363 (2008)
15. Haddadin, S., Belder, R., Albu-Schäffer, A.: Dynamic motion planning for robots in partially unknown environments. In: IFAC World Congress (IFAC 2011), Milano, Italy, vol. 18 (2011)
16. Haddadin, S., Haddadin, S., Khoury, A., Rokahr, T., Parusel, S., Burgkart, R., Bicchi, A., Albu-Schäffer, A.: On making robots understand safety: Embedding injury knowledge into control. *Int. J. of Robotics Research* 31, 1578–1602 (2012)
17. Haddadin, S., Suppa, M., Fuchs, S., Bodenmüller, T., Albu-Schäffer, A., Hirzinger, G.: Towards the robotic co-worker. In: International Symposium on Robotics Research (ISRR 2009), Lucerne, Switzerland (2009)
18. Haddadin, S., Urbanek, H., Parusel, S., Burschka, D., Roßmann, J., Albu-Schäffer, A., Hirzinger, G.: Realtime reactive motion generation based on variable attractor dynamics and shaped velocities. In: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2010), Taipei, Taiwan, pp. 3109–3116 (2010)
19. Hogan, N.: Impedance control: An approach to manipulation: Part I - theory, Part II - implementation, Part III - applications. *Journal of Dynamic Systems, Measurement and Control* 107, 1–24 (1985)
20. Ott, C.: Cartesian Impedance Control of Redundant and Flexible-Joint Robots. Springer Publishing Company, Incorporated (2008)
21. Park, J., Haddadin, S., Song, J., Albu-Schäffer, A.: Designing optimally safe robot surface properties for minimizing the stress characteristics of human-robot collisions. In: IEEE Int. Conf. on Robotics and Automation (ICRA 2011), Shanghai, China, pp. 5413–5420 (2011)
22. Parusel, S., Haddadin, S., Albu-Sch, A.: Modular state-based behavior control for safe human-robot interaction: A lightweight control architecture for a lightweight robot. In: IEEE Int. Conf. on Robotics and Automation (ICRA 2011), Shanghai, China, pp. 4298–4305 (2011)
23. Stemmer, A., Albu-Schäffer, A., Hirzinger, G.: An analytical method for the planning of robust assembly tasks of complex shaped planar parts. In: Int. Conf. on Robotics and Automation (ICRA 2007), Rome, Italy, pp. 317–323 (2007)
24. Vogel, J., Haddadin, S., Simeral, J.D., Stavisky, D., Bacher, S.D., Hochberg, L.R., Donoghue, J.P., van der Smagt, P.: Continuous control of the DLR Lightweight Robot III by a human with tetraplegia using the BrainGate2 neural interface system. In: International Symposium on Experimental Robotics (ISER 2010), Dehli, India (2010)
25. Yang, C., Gowrishankar, G., Haddadin, S., Parusel, S., Albu-Schäffer, A., Burdet, E.: Human like adaptation of force and impedance in stable and unstable interactions. Accepted: *IEEE Transactions on Robotics* (2011)
26. Zollo, L., Siciliano, B., De Luca, A., Guglielmelli, E., Dario, P.: Compliance control for an anthropomorphic robot with elastic joints: Theory and experiments. *ASME Journal of Dynamic Systems, Measurements and Control* (127), 321–328 (2005)

Understanding Functional Resonance through a Federation of Models: Preliminary Findings of an Avionics Case Study

Célia Martinie¹, Philippe Palanque¹, Martina Ragosta^{1,2}, Mark Alexander Sujan³,
David Navarre¹, and Alberto Pasquini²

¹ Institute of Research in Informatics of Toulouse (IRIT), University Paul Sabatier
118, route de Narbonne, 31062 Toulouse cedex 9, France

² DeepBlue S.r.l Consulting and Research, Piazza Buenos Aires 20, 00198 Roma, Italy

³ Warwick Medical School, University of Warwick, Coventry CV4 7AL, UK
{martinie, palanque, ragosta, navarre}@irit.fr,
{martina.ragosta, alberto.pasquini}@dblue.it,
m-a.sujan@warwick.ac.uk

Abstract. FRAM has been proposed as a method for the analysis of complex socio-technical systems, which may be able to overcome the limitations of traditional methods that focus on simple cause and effect relationships. FRAM on its own may be most useful for modeling the system at a high level of abstraction. There is less evidence about its utility for modeling interactions at greater levels of detail. We applied different modeling approaches to investigate situations that may give rise to functional resonance in an avionics case study. FRAM was used to model higher-level dependencies, HAMSTERS was used to provide a deeper understanding of human functions, and ICO-Petshop was used to model technical system functions. The paper describes preliminary results of the application of this federation of models, and highlights potential benefits as well as challenges that may have to be overcome.

Keywords: Modeling approaches, Avionics, Socio-technical systems.

1 Introduction

The causality of accidents in modern transportation systems may be difficult to determine. Investigations of past accidents and incidents have led to the development of improved system defences, which have significantly reduced the incidence of fatal accidents. When accidents occur they tend to exhibit complex causalities. Reason [1] referred to such accidents in modern well-defended systems as organizational accidents. These accidents are typically multi-faceted, and they may involve unexpected interactions or unforeseen propagation of failures [2].

In order to deal with the characteristics of modern transportation systems as well as other industrial safety-critical systems, paradigm changes to the existing safety engineering approaches have been proposed [3, 4]. Proponents of resilience engineering,

for example, have suggested regarding safety not simply as the absence of accidents, but rather as the ability to succeed under varying conditions [5].

The concept of functional resonance, developed within the resilience-engineering paradigm, describes accidents as the detectable “signal” that emerges from the unintended interaction of everyday variability. The Functional Resonance Analysis Method (FRAM) [6] is a corresponding modeling approach that has been put forward as a novel way of modeling and understanding the behavior of complex systems. FRAM has been used in a number of contexts including air traffic management [7], railway traffic management [8], healthcare [9] and financial services [10].

The functional description used by FRAM may be particularly useful for describing and analyzing systems at higher levels of abstraction. There is relatively little empirical evidence to demonstrate the application of FRAM to the detailed analysis of complex systems at different levels of abstraction. A possible strength of FRAM may be to make explicit the link between task-based and technical system descriptions, resulting in a federation of different modeling approaches.

The aim of this paper is to explore whether and how such a federation of different models can provide greater understanding of functional resonance in a real-world scenario. Section 2 provides a brief description of the avionics case study. Section 3 describes the learning generated through the application of FRAM. Sections 4 and 5, respectively, describe the learning generated from the application of a task-based modeling approach (HAMSTERS) and a petri-net based approach (ICO). Section 6 integrates and discusses the findings generated by this federation of models. Section 7 provides conclusions and suggestions for future research.

2 Weather Radar Interactive System

Weather radar (WXR) is an application currently deployed in many cockpits of commercial aircrafts. It provides support to pilots’ activities by increasing their awareness of meteorological phenomena during the flight journey, allowing them to determine if they may have to request a trajectory change, in order to avoid storms or precipitations for example. Annex 1 shows, on the cockpit of the Airbus A380, the distribution of various components dealing with weather radar.

Fig. 1 presents a screenshot of the weather radar control panel, used to operate the weather radar application. This panel provides two functionalities to the crew. The first one is dedicated to the mode selection of weather radar and provides information about status of the radar, in order to ensure that the weather radar can be set up correctly. The operation of changing from one mode to another can be performed in the upper part of the panel.

The second functionality, available in the lower part of the window, is dedicated to the adjustment of the weather radar orientation (Tilt angle). This can be done in an automatic way or manually (Auto/manual buttons). Additionally, a stabilization function aims to keep the radar beam stable even in case of turbulences. The right-hand part of Fig. 1 presents an image of the controls used to configure radar display, particularly to set up the range scale (right-hand side knob with ranges 20, 40, ... nautical miles).



Fig. 1. Image of a) the weather radar control panel b) of the radar display manipulation

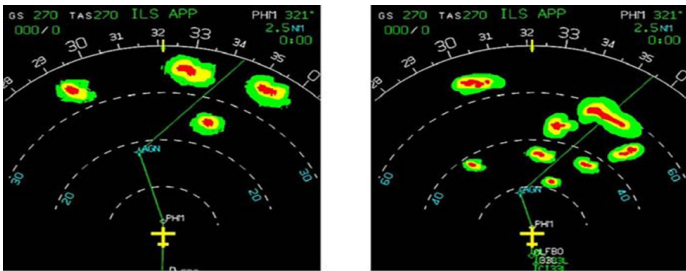


Fig. 2. Screenshot of weather radar displays

Fig. 2 shows screenshots of weather radar displays according to two different range scales (40 NM for the left display and 80 NM for the right display). Spots in the middle of the images show the current position, importance and size of the clouds.

The next three sections describe the learning for safety analysis generated by using complementary modeling approaches combined as a federation of models.

3 Functional Representation - FRAM

The safety analysis using FRAM is based on a functional representation of the system. Each function is described using six aspects - TROPIC (Time, Resource, Output, Precondition, Input, Control). The analysis using FRAM aims to investigate how the variability of the output of functions may propagate through the system, and how this propagation of variability may contribute to situations of functional resonance.

Fig. 3 graphically illustrates the functions identified for this case study. FRAM does not explicitly differentiate between the actors that perform a function. In the figure we have included an explicit representation of actors through the use of different levels of grey. Human functions are represented in light grey (continuous line for pilot functions while dotted for the air traffic controller). Functions performed by technical systems are represented in dark grey. Interactive functions are represented in medium grey. The functional description is hierarchical, so that functions can be

represented at higher levels of abstraction or with greater detail as required. For example, in Fig. 3 and Fig. 4 there is a function “Check weather conditions”, which is an abstraction of several lower-level functions (not represented in Fig. 3 above due to space constraints). This “macro” function includes system, human and interactive functions. Such an abstraction provides support for the representation of a larger number of functions while keeping the graphical model representation understandable.

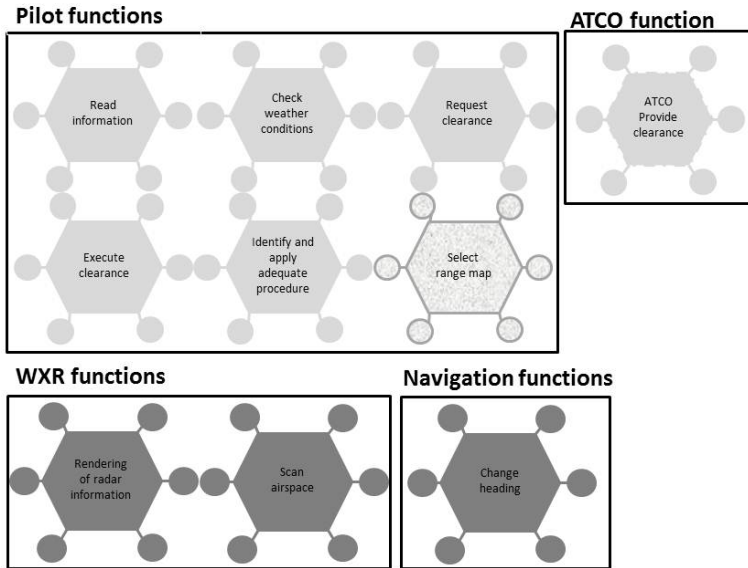


Fig. 3. Excerpt of FRAM functions for the weather radar socio-technical system

The FRAM instantiation shown in Fig. 4 describes a scenario where the pilot receives a clearance from the ATCO to change heading. The pilot checks the current weather situation and realizes that weather perturbations require a different route. Subsequently, the pilot requests a new clearance, which is eventually provided by the ATCO (who has to handle the impact of the refusal to implement the clearance on potential conflicts). In this scenario, the provision of the clearance also defines the timing parameter for executing the clearance, including checking the weather situation. Prior to executing a clearance the crew should check the weather conditions hence the latter function is a precondition for the former.

This simplified FRAM representation allows us to reason about the propagation of variability. For example, the provision of the clearance could vary in terms of its timing aspect, i.e. it could be provided late (for instance according to complex conflicts in the sector). In this case, there is less than adequate time available to execute the clearance. In such a situation, there may be a trade-off between efficiency and thoroughness in such a way that the weather check may be omitted (i.e. the precondition between execute clearance and check weather conditions) in order to save time.

This would lead to an execution of the clearance, rather than to the more appropriate request for a new clearance.

The functions provide clearance and request clearance use a shared resource, i.e. the communication link, which typically has limited bandwidth. A variation in the availability of this resource will again have implications for the timing of the functions. So, the assessment based on the FRAM representation suggests that variability due to timing and resource aspects may lead to potentially hazardous situations. The limitation of this approach is that without further models of human and technological systems behaviors, it is very difficult to explore and explain this potentially hazardous situation further. The next two sections will describe examples of such complementary modeling approaches that together may provide greater analytical power especially exhibiting quantitative information.

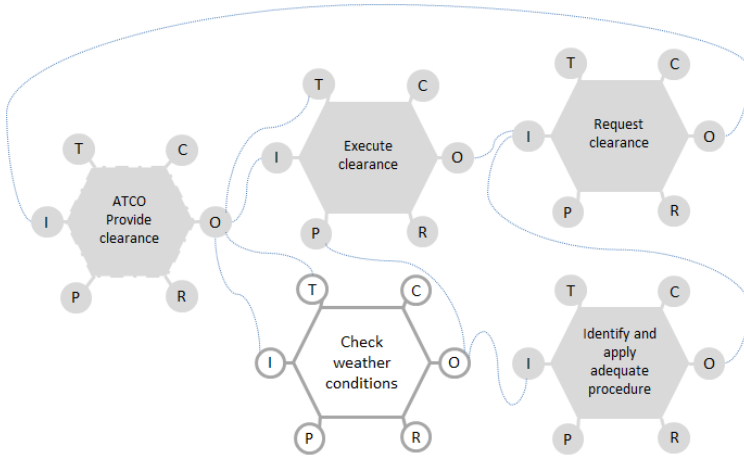


Fig. 4. FRAM instantiation (change heading clearance not feasible due to weather conditions)

4 Representing Operators’ Tasks Using HAMSTERS

HAMSTERS¹ is a tool-supported graphical task-modeling notation aiming at representing human activities in a hierarchical and ordered way. Goals can be decomposed into sub-goals, which can in turn be decomposed into activities, and the output of this decomposition is a graphical tree of nodes. Nodes can be tasks or temporal operators.

Tasks can be of several types (as illustrated in Fig. 5) and contain information such as a name, information details, critical level... Only the high-level task type are presented here (due to space constraints) but they are further refined (for instance the cognitive tasks can be refined in Analysis and Decision tasks [1]).

¹ <http://www.irit.fr/recherches/ICS/software/hamsters/index.html>










Task type	Icons in HAMSTERS task model
Abstract task	 Abstract task
System task	 System task
User task	 User task  Perceptive task  Cognitive task  Motor task
Interactive task	 Interactive input task  Interactive output task  Interactive input output task

Fig. 5. High-level task types in HAMSTERS

Temporal operators are used to represent temporal relationships between sub-goals and between activities. Some of them are represented in the task models below. Main ones are >> for sequence, ||| for concurrent, >| for interruptions and [] for exclusive choice.

Tasks can also be tagged by temporal properties to indicate whether or not they are iterative, optional or both [1]. Composition and structuration mechanisms have been introduced in order to provide support for description of complex activities [11]. One main element of these mechanisms is subroutine. A subroutine is a group of activities that a user performs several times possibly in different contexts and which might exhibit different types of information flows. A subroutine can be represented as a task model and a task model can use a subroutine to refer to a set of activities. This element of notation enables the distribution of large amount of tasks across different task models and factorization of the number of tasks.

HAMSTERS also provides support for representing how particular objects (data, information, knowledge ...) are related to particular tasks.

Fig. 6 illustrates the three relationships (input, output or input/output) between objects and tasks that can be expressed with HAMSTERS notation. Objects may be needed as an input to accomplish a particular task (as illustrated in Fig. 6a) by the incoming arrow). Particular tasks may generate an object or modify it (as illustrated in Fig. 6b and 6c)). According to the case study, the pilot has two main goals: “Keep awareness of weather situation” which includes the sub-goal “Checking weather conditions” (Fig. 7) and “Change heading” (not detailed here but involved in the execution of the considered clearance).

The task model in Fig. 7 represents crew activities performed in order to check weather conditions. At the higher level of the tree, there is an iterative activity (circular arrow symbol) to “detect weather targets” that is interrupted (operator >|) by a cognitive task “mental model of current weather map is built”.

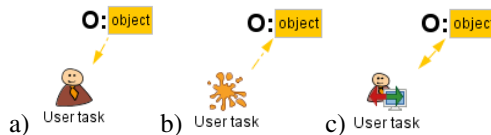


Fig. 6. Relationships between tasks and objects in HAMSTERS

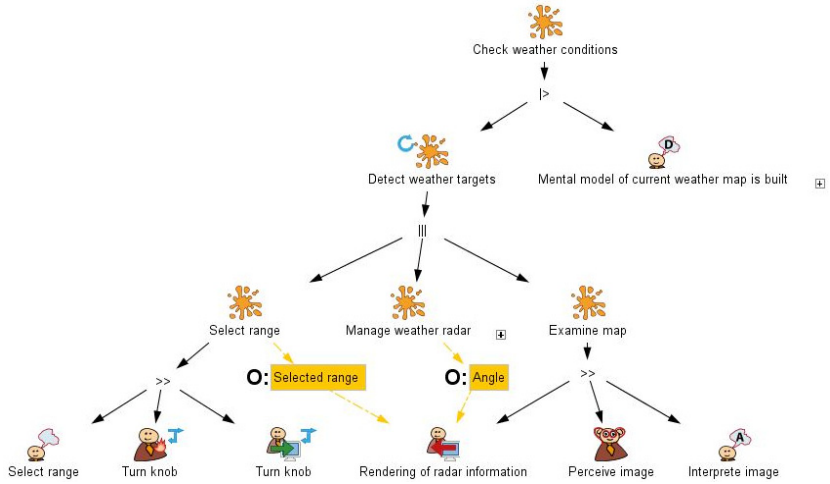


Fig. 7. HAMSTERS task model of the “Check weather conditions” goal

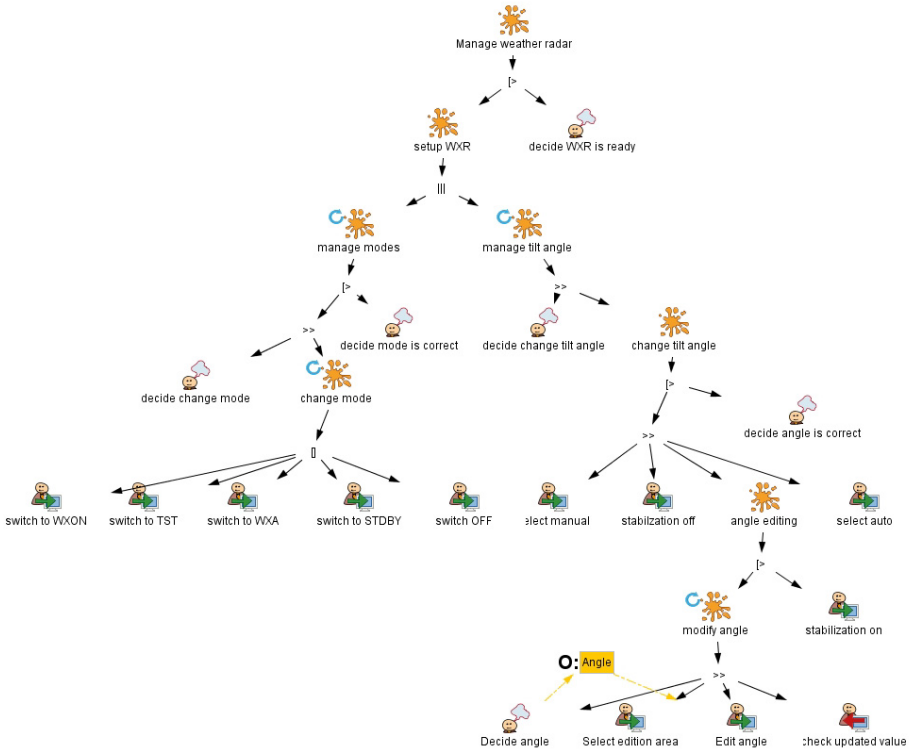


Fig. 8. HAMSTERS task model of the subroutine “Manage weather radar” task

Other human tasks include perception (task “Perceive image”) and motor (task “Turn knob”). Connection between crew’s activities and cockpit functions is made through interactive tasks (as input “Turn knob” and output “Rendering of radar information”). The time required for performing the latter heavily depends on the radar type. Such behavioral aspects of systems can be modeled using ICO notation and Pet-Shop tool as detailed in section 5. The task “Manage weather radar” is a subroutine task detailed in Fig. 8 and is performed after selecting a range and before analyzing the image produced by the weather radar. This task model corresponds to the manipulation of the user interface presented in Fig. 1 a). From these models we can see that the tasks to be performed in order to check weather conditions in a given direction are rather complex. The time required to perform them depends on 3 elements: the operator’s performance in terms of motor movements, perception and cognitive processing. Human performance models such as the one proposed in [12] can be used to assess difficulties and delays but the overall performance of the socio-technical system involves interaction and system execution times. Next section proposes a modeling approach for representing these two aspects while performance issues are presented in section 6.

5 Representing Technical Systems Using ICO Models

ICO [13] is used in this case study to model behavioral aspects of the system subpart of the interactive cockpit applications dealing with the weather radar. The following sections detail two models representing the interaction for controlling the weather radar parameters.

Mode Selection and Tilt Angle Setting

The first model presented here describes how it is possible to handle the weather radar configuration of both its mode and its tilt angle. **Fig. 1** shows the interactive means provided to the user to:

- Switch between the five available modes (upper part of the figure) using radio buttons (the five modes being WXON to activate the weather radar detection, OFF to switch it off, TST to trigger a hardware checkup, STDBY to switch it on for test only and WXA to focus detection on alerts).
- Select the tilt angle control mode (lower part of the figure) amongst three modes (fully automatic, manual with automatic stabilization and manual selection of the tilt angle).

Fig. 9 presents the description of the behavior of this part of the interactive cockpit using the ICO formal description technique and may be divided into two parts.

- The Petri net in the upper part handles events received from the 5 radio buttons. The current selection (an integer value from 1 to 5) is carried by the token stored in `MODE_SELECTION` place and corresponds to one of the possible radio buttons (OFF, STDBY, TST, WXON, WXA). The token is modified by the transitions (new_ms = 3 for instance) using variables on the incoming and outgoing arcs as formal parameters of the transitions. Each time the mode value is changed, the equipment part (represented by the variable wxr within the token) is set up accordingly.
- The Petri net in the lower part handles events from the four buttons and the text field (modify tilt angle). Interacting with these buttons changes the state of the

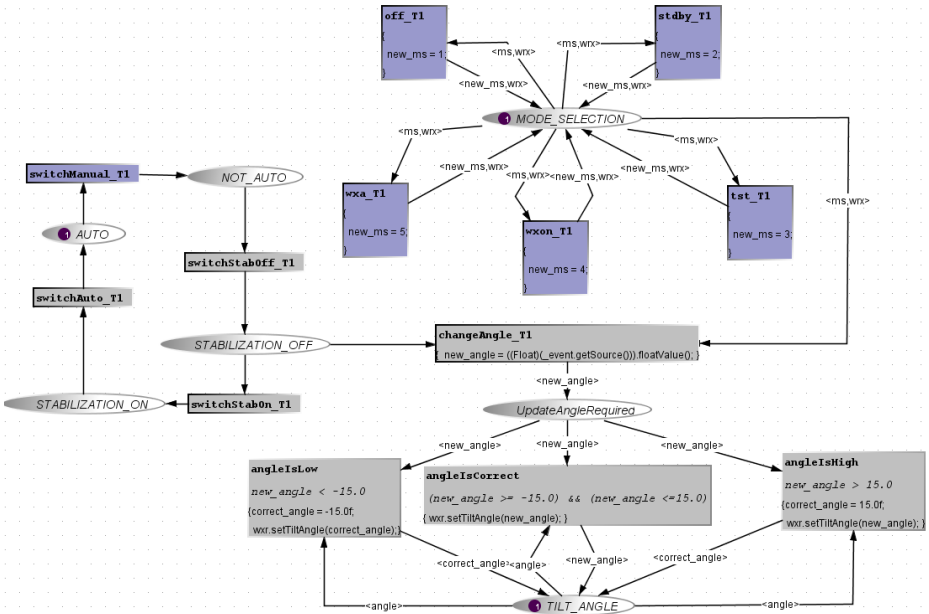


Fig. 9. Behavior of the WRX mode selection and tilt angle setting

application. In the current state, this part of the application is in the state fully automatic (a token is in AUTO place). To reach the state where the text field is available for the angle modification, it is necessary to bring the token to the place STABILIZATION_OFF by successively fire the two transitions switchManual_T1 and switchStabOff_T1 (by using the two buttons MANUAL and OFF represented by Fig. 1), making transition change_Angle_T1 available. The selected angle must belong to the correct range (-15 to 15), controlled by the three transitions angleIsLow, angleIsCorrect and angleIsHigh. When checked, the wxr equipment tilt angle is modified, represented by the method called wxr.setTiltangle.

Range Selection

The setting of the range detection of the weather radar is done using a FCU physical knob (see Fig. 1b) by switching between 6 values (from 1 to 6). Each time the value is set an event is raised (holding this value) by the knob and received by a dedicated part of the

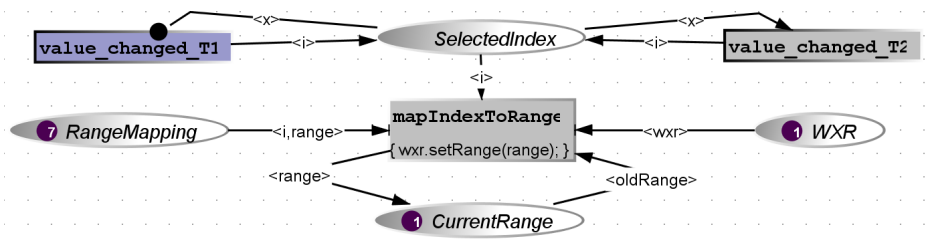


Fig. 10. Behavior of the range selection

cockpit application. This part of the application is represented by the model of **Fig. 10** that maps the value (form 1 to 6) into a range value that is sent to the WRX equipment.

The event is received and the selected value is extracted by one of the two transitions called valueChanged_T1 and valueChanged_T2. The place RangeMapping contains the mapping between a value and the corresponding range (for instance 1 corresponds to range 10, 2 to 20...). Finally, the wxr equipment range is set with the selected range by the firing of transition mapIndexToRange.

6 Time and Performance Aspects

To allow performance assessment we have to address timing issues at three levels: the operator side using the task models presented in section 4, the system side exploiting the ICO behavior models in section 5, the interaction side related to the graphical interface described also in ICOs in section 5.

Operators' Performance from Task Models

To qualitatively evaluate the performance of the weather radar graphical interface, we first restricted the study to the interaction with the weather radar control panel (see Fig. 1a), which is handled using a trackball (the other part being handled using the knob). One of the evaluation approaches used in human factors domain is based on Fitts's law [14] which is suitable for assessing motor movements. Fitts's law is presented in Formula (1) representing an index of difficulty for reaching a target (of a given size) from a given distance. Movement time for a user to access a target depends on width of the target (W) and the distance between the start point of the pointer and the center of target (A).

$$MT = a + b \log_2(1+2A/W) \tag{1}$$

For predicting movement time on the systems under consideration constants are set as follows: a=0 and b=100ms (mean value for users).

Fig. 11 presents the set of interactive widgets used within the weather radar control panel. For each widget, it provides a short name used for the following tables and the size used as the width for the Fitts's law (we use the minimum value between the width and the height to provide the assessment of the maximum difficulty to reach the considered widget).

Fig. 12 provides the distances from the center to each widget and between each widget. These distances are used to apply the Fitts's law when reaching a widget with a start point that can be the center of the control panel or any widget.

	radio button off	radio button stdby	radio button tst	radio button wxon	radio button wxa	button Auto	button Manual	button ON	button OFF	text field angle
short name	r1	r2	r3	r4	r5	b1	b2	b3	b4	t1
min size	18	18	18	18	18	31	31	31	31	26

Fig. 11. Interactive widgets width used for the Fitts's law application

	r1	r2	r3	r4	r5	b1	b2	b3	b4	t1
c	104	130	115	100	87	77	17	128	104	132

	r1	r2	r3	r4	r5	b1	b2	b3	b4	t1
c	110	119	114	108	103	77	32	97	89	105

Fig. 12. Distance from the control panel center a) Temporal values (in ms) for user interaction using Fitts's b)

Model	Transition	Duration (ms)
WXR control panel model	Off_T1	500
	Stdby_T1	200
	Wxa_T1	500
	Wxon_T1	1000
	Tst_T1	1000
	angleIsLow	2000-4000
	angleIsCorrect	2000-4000
	angleIsHigh	2000-4000
Range selection model	mapIndexToRange	200

Fig. 13. Delays introduced by interaction

In addition to these motor values cognitive and perceptive values have to be used in order to cover all the elements of the task models. From [12] we know that the mean time for performing a comparison at the cognitive level is 100ms (ranging from 25ms to 170ms) while eye perception mean is 100mn too (ranging from 50ms to 200ms).

Weather Radar System Time (Associated to ICO Models)

In the ICO Petri net dialect, time is directly related to transition, which invokes services from the weather radar system (this is the case for transition off_T1 on Fig. 9 which switches off the equipment). The duration of each invocation is presented on Fig 13 (each value is coarse grain and depends on the type of weather radar). The 2000-4000ms value corresponds to the time required by the weather radar to scan the airspace in front of the aircraft (two or three scans are needed to get a reliable image).

Using the task models in Fig. 7 and Fig. 8 and the values above we can estimate the overall performance of the crew to perform the “check weather condition” task. The overall time cannot be less than 30 seconds provided that several ranges have to be checked in turn. Going back to FRAM model presented on Fig. 4 the function “check weather condition” is a strong bottleneck and influences the entire socio technical system.

7 Conclusion

The paper outlined how a federation of three complementary modeling paradigms could be a useful approach in order to explore situations of functional resonance within socio-technical systems. FRAM provided a high-level view of possible dependencies in the system under consideration. These dependencies were then further explored using HAMSTERS for human activities, and ICO-Petshop for technical systems covering both interaction techniques on the user interfaces and the underlying hardware and software systems. The analysis presented in this paper represents a first step and the results are preliminary. A possible limitation of the approach is that there is no clear algorithm for how the three models can interact. At present, this relies on the skill of the analyst. This may pose problems in the analysis of large systems, where a greater level of tool support may be required for the analysis but it is important to note that most of the modeling activities are supported by tools and that performance evaluation techniques are partially available as for ICOs with Petri nets theory [15] and HAMSTERS with dedicated tools presented in [11]. Future research should investigate the generalizability of this case study to larger systems. The possibility of formalizing the interaction between the models at the different levels of analysis should be explored further. Lastly, variability and resonance

can also occur through system failure occurs or operators errors. We aim at integrating previous work we have done in the area of systems reconfiguration [16] and systematic account for human error using task models [17] to address variability for all the components of the socio-technical system.

Acknowledgements. This work is partly funded by Eurocontrol research network HALA! on Higher Automation Levels in Aviation and SPAD project (System Performance under Automation Degradation).

References

1. Reason, J.: *Managing the risks of organizational accidents*. Ashgate (1997)
2. Perrow, C.: *Normal Accidents*. Basic Books (1984)
3. Leveson, N.: *Engineering a safer world*. MIT Press (2011)
4. Hollnagel, E., Woods, D.D., Leveson, N.: *Resilience Engineering: Concepts and Precepts*. Ashgate (2006)
5. Hollnagel, E.: Prologue: The Scope of Resilience Engineering. In: Hollnagel, E., et al. (eds.) *Resilience Engineering in Practice*. Ashgate (2010)
6. Hollnagel, E.: FRAM: The functional resonance analysis method for modelling complex socio-technical systems. Ashgate, Farnham (2012)
7. Herrera, I.A., Woltjer, R.: Comparing a multi-linear (STEP) and systemic (FRAM) method for accident analysis. *Reliability Engineering & System Safety* 95, 1269–1275 (2010)
8. Belmonte, F., Schön, W., Heurley, L., et al.: Interdisciplinary safety analysis of complex socio-technical systems based on the functional resonance accident model: an application to railway traffic supervision. *Reliability Engineering & System Safety* 96, 237–249 (2011)
9. Sujan, M.-A., Felici, M.: Combining Failure Mode and Functional Resonance Analyses in Healthcare Settings. In: Ortmeier, F., Lipaczewski, M. (eds.) *SAFECOMP 2012*. LNCS, vol. 7612, pp. 364–375. Springer, Heidelberg (2012)
10. Sundström, G.A., Hollnagel, E.: Governance and control of financial systems. A resilience engineering approach. Ashgate, Aldershot (2011)
11. Martinie, C., Palanque, P., Winckler, M.: Structuring and Composition Mechanisms to Address Scalability Issues in Task Models. In: Campos, P., Graham, N., Jorge, J., Nunes, N., Palanque, P., Winckler, M. (eds.) *INTERACT 2011, Part III*. LNCS, vol. 6948, pp. 589–609. Springer, Heidelberg (2011)
12. Card, S., Moran, T., Newell, A.: The Model Human Processor: An Engineering Model of Human Performance. In: *Handbook of Perception and Human Performance*, pp. 1–35 (1986)
13. Navarre, D., Palanque, P., Ladry, J.-F., Barboni, E.: ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.* 16(4), , Article 18, 56 pages (2009)
14. Fitts, P.M.: The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology* 47, 381–391 (1954)
15. Ajmone Marsan, M., Balbo, G., Conte, C., Donatelli, S., Franceschinis, G.: *Modelling with generalized stochastic Petri nets*. Wiley (1995)
16. Navarre, D., Palanque, P., Basnyat, S.: A Formal Approach for User Interaction Reconfiguration of Safety Critical Interactive Systems. In: Harrison, M.D., Sujan, M.-A. (eds.) *SAFECOMP 2008*. LNCS, vol. 5219, pp. 373–386. Springer, Heidelberg (2008)
17. Palanque, P., Basnyat, S.: Task Patterns for taking into account in an efficient and systematic way both standard and erroneous user behaviours. In: *HESSD 2004 6th Int. Working Conference on Human Error, Safety and System Development*, pp. 109–130 (2004)

Model-Based Development of the Generic PCA Infusion Pump User Interface Prototype in PVS

Paolo Masci¹, Anaheed Ayoub², Paul Curzon¹,
Insup Lee², Oleg Sokolsky², and Harold Thimbleby³

¹ Queen Mary University of London, UK
{paolo.masci,paul.curzon}@eecs.qmul.ac.uk

² University of Pennsylvania, PA, USA
{anaheed,lee,sokolsky}@cis.upen.edu

³ Swansea University, UK
harold@thimbleby.net

Abstract. A realistic user interface is rigorously developed for the US Food and Drug Administration (FDA) Generic Patient Controlled Analgesia (GPCA) pump prototype. The GPCA pump prototype is intended as a realistic workbench for trialling development methods and techniques for improving the safety of such devices. A model-based approach based on the use of formal methods is illustrated and implemented within the Prototype Verification System (PVS) verification system. The user interface behaviour is formally specified as an executable PVS model. The specification is verified with the PVS theorem prover against relevant safety requirements provided by the FDA for the GPCA pump. The same specification is automatically translated into executable code through the PVS code generator, and hence a high fidelity prototype is then developed that incorporates the generated executable code.

Keywords: Formal methods, Model-based development, Medical devices, User interface prototyping.

1 Introduction and Motivation

Infusion pumps are medical devices used to deliver drugs to patients at precise rates and in specific amounts. The current infusion pumps incorporate sophisticated software, of around tens of thousands of lines of program code [9]. This complexity may make infusion pumps flexible and configurable, but it introduces new risks as software correctness is hard to verify. Traditional manual verification and validation activities based on manual inspection, code walkthroughs and testing are insufficient for catching bugs and design errors in such complex software. Unfortunately there are currently no widely accepted techniques for development and verification of software for medical devices, nor standard guidelines to ensure that a device meets given safety requirements [8].

Numerous adverse events have been reported that are associated with infusion pumps. Reports from the US Food and Drug Administration (FDA) show that

some of these incidents are due to use errors and software failures caused by poor software design [5]. Because of this, several device recalls have been issued: for instance, 87 models of infusion pump, affecting all infusion pump manufacturers, were recalled over 2005 to 2009 in the US [5].

The FDA is promoting the development of so-called Generic Infusion Pump (GIP) models and prototypes as a way to demonstrate how rigorous development methods can substantially improve confidence in the correctness of software. For instance, in [1], the FDA presents a research prototype for generic design of Patient Controlled Analgesia (PCA) pumps, called the Generic PCA (GPCA) pump. We explain PCA pumps more fully below.

The GPCA itself is not yet a real medical device. However, because its functionalities and details closely resemble those of a real medical device, it can be used as a realistic workbench. Successful application of methods and tools to the GPCA prototype should indicate that they are viable for commercial devices.

The importance of user interface design is well understood by regulators [21]. However, hardly any concrete examples of model-based development of user interfaces have been explored that take account of human factors or human factors engineering. In our previous work, we illustrated how verification tools could be used to verify the design of commercial infusion pump user interfaces against properties that capture human factors concerns [4, 6, 13, 14] and safety requirements [12]: potential issues were identified precisely, and verified design solutions that could *fix* the identified issues were presented. This work builds on our previous work, and extends it by introducing a model-based development approach for rapid prototyping of medical device user interfaces that are verified against given safety requirements. The approach presented in this paper is illustrated through the development of core parts of the user interface of the GPCA.

Contributions. The main contribution of this paper is the detailed model-based development of a data entry system for the GPCA user interface within Prototype Verification System, PVS [18]. The specification of the data entry system of the GPCA user interface incorporates safety features that can mitigate use errors, and the specification is formally verified against safety requirements provided by the FDA within PVS, a standard system commonly used for this purpose. The verified model is then automatically transformed into executable code through the PVS code generator, and the generated code is then incorporated in a prototype that can be executed.

2 Related Work

In this paper, the model-based approach is implemented using the Prototype Verification System (PVS) [18]. PVS is a state-of-the-art verification tool that provides an expressive specification language based on higher-order logic, a language mechanism for theory interpretation [19], a verification engine based on automated theorem proving, and a code generator for automatic translation of PVS specifications into executable code [23].

PVS is only one approach of course, and other tools could have been used to develop the prototype. Our choice was guided by pragmatics linked to best

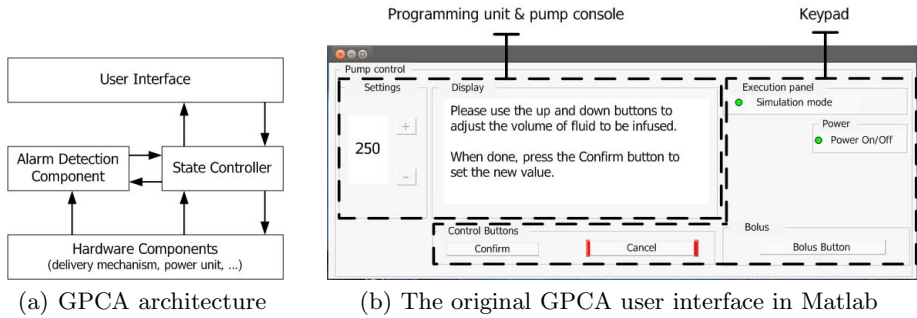


Fig. 1. Schematic of the FDA’s Generic PCA infusion pump prototype

current development practices — the need to specify safety requirements independently, expressiveness of the specification language (here, PVS), and automated code generation from verified specifications. In [10], different tools are used to generate a prototype of the GPCA pump controller with a model-based approach: safety requirements are formalised as properties of a Uppaal model of the GPCA controller, and the Times code generator is used for translating the model into platform-independent C code. In [27], a model of the Generic Insulin Infusion pump controller are developed in Event-B using the Rodin platform. The model is verified against selected safety requirements related to timing issues. The development of a prototype from the verified model was not in the scope of that work. In [3], model-based development is used to generate a runtime software monitor that validates the behaviour of an insulin infusion pump against safety requirements. Petri Nets are used to specify the behaviour of the monitor, and then the specification is manually translated into Java code.

No prior work addresses model-based development of realistic user interfaces for infusion pumps.

3 The Generic PCA (GPCA) Pump

PCA pumps are a class of infusion pump used to administer pain relief medication. They are employed for self-administration where a patient is able to request pain relief in controlled amounts when they need it. The patient interacts with the PCA pump using a single button, which is used to request additional pre-defined doses of drug. The intended infusion parameters are programmed in advance by clinicians. In the current generation of infusion pumps, clinicians program infusion parameters by interacting with buttons on the user interface.

The FDA has developed a Matlab Simulink/Stateflow model of the Generic PCA (GPCA) pump that captures the core functionalities of PCA pumps in general. The GPCA model has a layered architecture (see Figure 1(a)). The top layer is the user interface, which presents the state of the infusion pump and allows users to program infusion parameters. The software controller is the middle layer in the architecture, and includes two components: a state controller and alarm detection component. The state controller drives the drug administration process and

supervises communication among the modules of the GPCA pump. The alarm detection component handles alarms and warnings. The lowest layer models the hardware components, such as the delivery mechanism (peristaltic motors and air bubble sensors, etc) and power unit (including the battery charger, etc).

The GPCA user interface, as provided with the original model [21], has the layout shown in Figure 1(b). The user interface includes the following elements: a *programming unit and pump console*, which renders information about the pump state and allows users to set infusion parameters; and a *keypad*, which allows users to send commands to the pump. Human factors were not considered when developing this original user interface [21], as the user interface was used primarily for development and debugging purposes.

3.1 GPCA Safety Requirements

The FDA has released an initial set of 97 GPCA safety requirements [1]. They are formulated in natural language, and grouped into 6 main categories: *infusion control*, which are dedicated to safety features and constraints that can mitigate hazards resulting from incorrectly specified infusion parameters (e.g., flow rate too high or too low); *user interface*, which describe constraints on user interface functionalities that can help avoid accidental modification of infusion parameters; *error handling*, which are dedicated critical alarming conditions; *drug error reduction*, which define drug library functionalities; *power and battery operations* and *system environment*, which are dedicated to constraints on operating conditions.

The GPCA safety requirements describe essential safety features and constraints that guarantee a minimum level of pump safety. The requirements were obtained by reasoning about mitigation actions that could contrast identified hazards associated with PCA pumps, as well as related causes of the identified hazards. For instance, an identified hazard of PCA pumps is overinfusion, and one of the causes is that the programmed flow rate is too high. A suggested mitigation for this hazard is to make the flow rate programmable within given rate bounds only. Starting from this suggested mitigation, corresponding GPCA safety requirements are then formulated that can help check the mitigation barrier in the pump.

The GPCA safety requirements were designed on the basis of a preliminary hazard analysis for the controller of the pump. We found that almost half of the requirements can be related to user interface functionalities, and correctly capture basic human factors concerns. However, a hazard analysis specifically addressing user interface functionalities is needed to cover a more complete set of aspects related to human factors. We are currently starting this hazard analysis. Some examples of safety features and constraints that are currently *not* considered in the GPCA safety requirements and can potentially make the user interface design safer follows.

Illegal keying sequences shall be blocked during interactive data entry. An illegal keying sequence is a sequence of key clicks resulting in an illegal value (e.g., a value out of range) or illegal number format (e.g., a number with two decimal

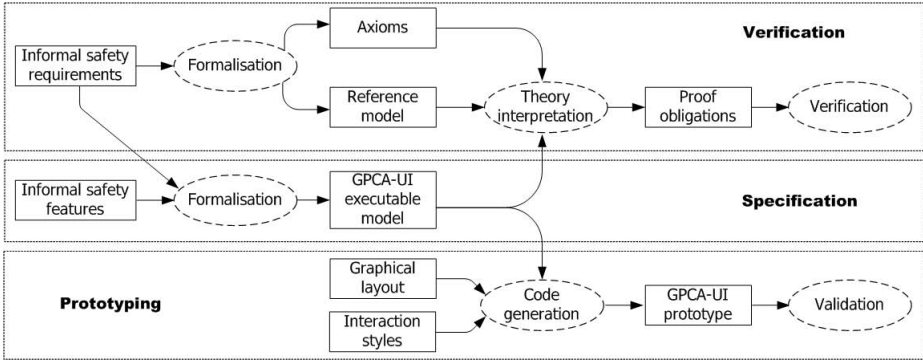


Fig. 2. The adopted model-based development approach

dots). Blocking an illegal keying sequence means that interaction is halted when a key click results in an illegal keying sequence, and feedback is provided to the user. Results presented in [4] and [25] show that this safety feature can create useful mitigation barriers against keying slip errors.

Numbers rendered on the display shall follow the ISMP rules [7]. The Institute for Safe Medical Practices (ISMP) promoted the adoption of two basic rules to design correctly formatted dosage values: leading zeros (e.g., 0.1 mL) are always required for fraction dose values; and trailing zeros (e.g., 1.0 mL) are always avoided. These rules are distilled from best practice (e.g., 1.0 may be misread as 10) and aim to reduce medication errors through standardised formatting and clear presentation.

Numbers rendered on displays shall follow the NPSA recommendations [26]. The UK National Patient Safety Agency (NPSA) recommends adherence to the following guidelines to facilitate a correct identification of dosage values, including: digits after the decimal point are rendered in smaller font size; the visual salience of the decimal point is increased; using “TallMan” lettering for names and units.

4 Development of the GPCA User Interface

A GPCA user interface (hereafter, GPCA-UI) prototype is now developed using a model-based approach. Within model-based development approaches, models are used as primary artefacts during the whole development cycle: they present a design specification that can be checked against given requirements, and then code generation techniques are used to transform the model into a concrete implementation for a specific platform. The adopted model-based development approach consists of the following phases (as shown in Figure 2):

Specification. A GPCA-UI model is specified that defines the interactive behaviour of the GPCA user interface. The specification is given in the form of an *executable model*, that is a model whose specification is rich enough that it can be systematically translated into a concrete code implementation.

Verification. The developed GPCA-UI model is verified against a formalisation of selected GPCA safety requirements. This is done through an approach based on *theory interpretation*. It is based on the idea of formalising safety requirements as axioms of an abstract model (which we call the *reference model*); proof obligations that need to be verified on the GPCA-UI model are automatically generated by mapping functionalities of the abstract model into functionalities of the GPCA-UI model; a formal verification is then performed to discharge the generated proof obligations. With this approach, safety requirements can be developed independently from the GPCA-UI model.

Prototyping. A user interface prototype is developed. The prototype incorporates software code automatically generated from the verified GPCA-UI model through code transformation. The prototype has a back-end that defines the functionalities of the GPCA-UI, and a front-end that defines the visual appearance of the GPCA-UI. The back-end is executed within a verified execution environment to ensure that correctness properties verified on the formal specification are preserved at run-time when executing the generated code. The front-end just renders information returned by the back-end. The prototype can be used for validation purposes. The prototype implementation in PVS is further elaborated in the following sub-sections.

4.1 Specification

The model is developed as a finite state machine. The state of the state machine defines information observable on the GPCA-UI (e.g., values shown on a display) and internal variables (e.g., values held by timers). State transitions of the state machine define interactive functionalities activated by the operator (e.g., button clicks) and internal events generated by the GPCA-UI (e.g., timer events).

The GPCA-UI model includes the typical functionalities provided by the current generation of commercial PCA pump user interfaces. Due to space limitations, only a qualitative description of the functionalities included in the model is provided here without going into the specific details of the PVS model. The full PVS model can be found at [2].

The GPCA-UI programming unit specifies the behaviour of a “5-key” number entry [4, 17], as widely used in commercial infusion pumps. A different choice could have been made (chevron keys, number pad, or others). Two functions (*up* and *down*) edit the entered value by an increment step. The increment step is proportional to the position of a cursor. Two functions (*left* and *right*) edit the position of the cursor. The accuracy of the entered value is limited to two decimal digits, and legal values are below between 0 and 99999. These limits reflect those of commercial PCA pumps. Whenever these limits are violated, interaction is halted and an alert message displayed.

The GPCA keypad specification defines the basic behaviour of typical commands made available to the operator to control the state of the pump: turn the pump on and off; start and stop an infusion. Additional functionalities not implemented in this first version of the model include: edit infusion parameters; view pump status; deliver an additional limited amount of drug upon demand.

The model developed includes a specification of the communication protocol with the GPCA controller developed by Kim *et al* in [10]. In the current version, the protocol specification includes the sequence of commands to boot-strap the pump controller.

4.2 Verification

Within the verification approach, safety requirements formulated in natural language are formalised as predicates (see Figure 2). These predicates define the functionalities of a logic-based model, which we call the *reference model*, which encapsulates the semantics of the safety requirements by construction. The reference model is used for the verification of the GPCA-UI model by means of a technique called *theory interpretation* [19], which is a verification approach based on the idea of establishing a mapping relation between an abstract model and a concrete model. The mapping relation is used to systematically translate properties that hold for the abstract model into proof obligations that need to be verified for the concrete model. In our case, the abstract model is the reference model, and the concrete model is the GPCA-UI model. Hence, safety requirements encapsulated in the specification of the reference model are systematically translated into proof obligations for the GPCA-UI model. Being able to discharge the generated proof obligations through formal proof is a demonstration that the GPCA-UI model meets the safety requirements. The GPCA-UI specification developed is then formally verified against the following GPCA requirements that are relevant to the data entry system.

GPCA 1.1.1. *The flow rate of the pump shall be programmable.*

GPCA 1.1.2. *At a minimum, the pump shall be able to deliver primary infusion at flows throughout the range f_{min} and f_{max} mL per hour.*

GPCA 1.3.1. *The volume to be infused settings shall cover the range from v_{min} to v_{max} mL.*

GPCA 1.3.2. *The user shall be able to set the volume to be infused in j mL increments for volumes below x mL.*

GPCA 1.3.3. *The user shall be able to set the volume to be infused in k mL increments for volumes above x mL.*

Example. Requirement 1.3.1 is formalised and verified to exemplify the verification approach. A logic expression is created by extracting the relevant concepts presented in the textual description: *VTBI settings range* (where VTBI means volume of drug to be infused), v_{min} and v_{max} . As shown in Listing 1.1, these concepts are used to define an uninterpreted predicate `vtbi_settings_range` in PVS higher-order logic, and two symbolic constants `v_min` and `v_max` of type non-negative real numbers. The state of the reference model is specified with a new uninterpreted type, `ui_state`.

Listing 1.1. Part of the Reference Model

```
ui_state: TYPE
vtbi_setting_range(vmin,vmax: noneg_real)(st:ui_state): boolean
vmin,vmax: noneg_real
```

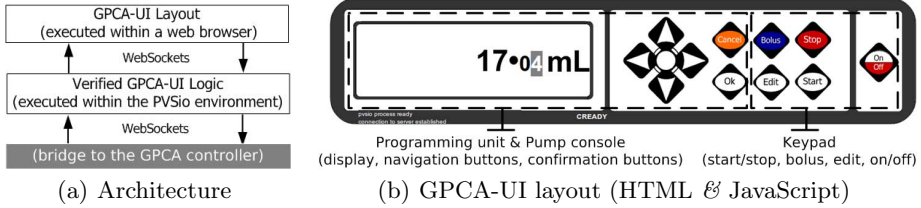



Fig. 3. The GPCA-UI prototype

The predicate and constants are then encapsulated in the reference model by formulating a property that always holds for the reference model (i.e., an *axiom*). In PVS, axioms are boolean expressions annotated with the `AXIOM` keyword. Axioms can be formalised in a way that facilitates proof by structural induction. For the considered example, the formalisation is as follows (see Listing 1.2). The initial state of the reference model (predicate `init?`) satisfies `vtbi_settings_range` (the *induction base*); given a state `st0` of the reference model that satisfies `vtbi_settings_range`, any state `st1` reachable from `st0` through a transition function (`trans`) of the reference model satisfies `vtbi_settings_range` (the *induction step*).

Listing 1.2. Axiom used to specify requirement R1

```

R1_Axiom: AXIOM
FORALL (st, st0, st1: ui_state):
  (init?(st) IMPLIES vtbi_settings_range(vmin, vmax)(st)) AND
  ((vtbi_settings_range(vmin, vmax)(st0) AND trans(st0, st1))
   IMPLIES vtbi_settings_range(vmin, vmax)(st1))
    
```

A relation is then defined that specifies how `vtbi_settings_range` is mapped into the GPCA-UI model. In this case, the relation maps `vtbi_settings_range` into a function that checks the `vtbi` range supported by the GPCA-UI model (the second `LAMBDA` function in Listing 1.3). Through this mapping, PVS is able to automatically generate proof obligations that must be verified on the GPCA-UI model in order to demonstrate compliance with the reference model (and, hence, show that the safety requirement is met). The syntax for specifying a theory interpretation in PVS is that of a PVS theory importing clause (keyword `IMPORTING` followed by the model name, `reference_model.th` in this case) with actual parameters specifying the mapping relation (a list of substitutions provided within double curly brackets). Listing 1.3 gives a snippet of the PVS theory interpretation specified for the considered requirement: it states that the uninterpreted state of the reference model (`ui_state`) is mapped onto the state of the GPCA-UI model (`gpcaui_state`). The uninterpreted predicate that recognises the initial state of the reference model (`init?`) is mapped into the interpreted predicate that recognises the initial GPCA-UI concrete model state (`gpcaui_init?`). The uninterpreted predicate that identifies the set of transitions of the reference model (`trans`) is mapped into a function that enumerates the transition functions of the GPCA-UI concrete model (the first `LAMBDA` expression in the specification snippet shown in Listing 1.3; in the expression, `st_prime` identifies the next state obtained after applying a transition function).

Listing 1.3. Theory interpretation

```

IMPORTING reference_model_th
{{ ui_state := gpcaui_state,
  init? := gpcaui_init?,
  trans := LAMBDA (st, st_prime: gpcaui_state):
    st_prime = click_up(st) OR %...
  vmin := 0, vmax := 99999,
  vtbi_settings_range
    := LAMBDA (vmin, vmax: nonneg_real)(st: gpcaui_state):
      vmin <= display(st) AND display(st) <= vmax
      AND vmin <= vtbi(st) AND vtbi(st) <= vmax }}

```

Given this theory interpretation, PVS automatically generates the proof obligation in Listing 1.4, which then needs to be discharged. The proof obligation requires we show that, for all reachable states, it is always true that the display and the VTBI range have values between `v_min` and `v_max` (0-99999 in this case).

Listing 1.4. Proof obligation

```

IMP_reference_model_th_R1_Axiom_TCC1: OBLIGATION
FORALL (st, st0, st1: gpcaui_state):
  (gpcaui_init?(st) IMPLIES
    0 <= st'display AND st'display <= 99999
    AND 0 <= st'vtbi AND st'vtbi <= 99999)
AND ((0 <= st0'display AND st0'display <= 99999 AND 0 <= st0'vtbi
  AND st0'vtbi <= 99999 AND st1 = click_up(st0) OR %...)
  IMPLIES 0 <= st1'display AND st1'display <= 99999
  AND 0 <= st1'vtbi AND st1'vtbi <= 99999);

```

The generated proof obligation can be discharged within the PVS theorem prover thanks to using implicit subtype constraints [24] declared for the `vtbi` type and the `display` type in the GPCA-UI model. (In PVS, implicit subtype constraints are made explicit by using the command `typepred`.) After making subtype constraints explicit, the proof can be completed in less than a second with `assert`, a predefined decision procedure of the PVS theorem prover that simplifies expressions using decision procedures for equalities and linear inequalities. Alternatively, PVS can perform the proof automatically in seconds with its command `grind`, a powerful predefined decision procedure that repeatedly applies definition expansion, propositional simplification, and type-appropriate decision procedures.

4.3 Prototyping

An interactive GPCA-UI prototype is now presented that incorporates the verified PVS specification. The utility of the prototype is that it allows validating the behaviour of the generated code, and verifying aspects of the UI that are not formalised in the specification (e.g., the guidelines illustrated in Section 3.1). Additionally, the prototype can be used by formal methods experts to engage with domain experts such as human factors specialists.

The GPCA-UI prototype can be downloaded at [2]. The prototype architecture is split into a front-end and a back-end, as shown in Figure 3(a). The front-end is deployed on a tablet, which makes it possible to do realistic interaction

with the buttons on the user interface. The back-end is deployed on a server with the PVSio [15] prototyping environment. Code automatically generated from the PVS specification is executed exclusively on the back-end within the Lisp execution environment of PVS. This gives us confidence that the safety requirements verified for the GPCA-UI specification are preserved when executing the Lisp code automatically generated from the verified GPCA-UI specification.

The design choices for the front-end and back-end are valid for the illustrative purpose of this work, that is to generate a realistic user interface for a research prototype from a verified model:

The GPCA-UI front-end is responsible for the visual appearance of the GPCA-UI. A “5-key” number entry layout based on navigation buttons has been chosen because it is widely used in the current generation of commercial PCA pumps. A different choice could have been made (chevron keys, number pad, or others). The front-end is executed within a web browser, which very conveniently allows using HTML code to render the GPCA-UI layout and using JavaScript to capture user interactions with buttons and translate them into function calls for the PVSio environment executed on the back-end. This translation from user actions to commands is performed on the basis of mappings between interactive areas of the GPCA-UI and function names in the PVS specification of the GPCA-UI. An example mapping that has been defined is the following: a button click of the *up* arrow key triggers a call to function `click_up` in the PVS specification. JavaScript is used to render the user interface state returned by PVSio: it renders numbers in the GPCA-UI display in a way that is compliant with the ISMP and NPSA recommendations given in Section 3.1. This can be validated through visual inspection. Note that the developed HTML and JavaScript code do not add new behaviours to the GPCA-UI — they are just used to send commands and render the state returned by the back-end.

The GPCA-UI back-end is responsible for the interactive behaviour of the GPCA-UI. The core of the back-end is the PVSio [15] prototyping environment. It provides an interactive command prompt that accepts higher-order logic expressions. The expressions are evaluated in the Lisp execution environment of PVS: Lisp code is generated on-demand, and then executed. A result is returned symbolically every time an expression is evaluated. The returned expression is a GPCA-UI model state, in this case. For instance, writing the expression `click_up(init)` in the PVSio command prompt results in the evaluation of function `click_up` of the GPCA-UI specification starting from state `init`. Lisp code is automatically generated, the function is executed, and a new state returned. A web-server presents the PVSio command prompt as a service of the GPCA-UI back-end. WebSockets, a standard protocol for bidirectional low-latency communication between two endpoints over a TCP connection, are used to enable communication between the front-end and the back-end.

5 Conclusions

Making medical devices safer involves a constructive dialogue among stakeholders (manufacturers, regulators, clinicians), and a verification approach based on

these generic models can help to make this dialogue precise, as well as having the advantages of being computerized and runnable.

We have presented a model-based development approach for building a realistic user interface for the GPCA pump prototype. Although the user interface is a research prototype and not a real medical device, the functionalities and level of detail used in the specification are very similar to those of commercial PCA pumps. Because of this, it is evident that the specification can be used as a realistic workbench, and the model-based developed approach used can in principle be used as part of the development of real medical device user interfaces.

The model-based approach incorporates several concepts promoted by medical device regulators and which should be directly applicable to the development of real medical devices. For instance, in [21] and [9], the FDA Office of Science and Engineering Lab (OSEL) engineers have promoted the formalisation of safety requirements as generic models that can be used for verification of real devices.

The model-based approach introduced here has some limitations that need to be considered and should be the subject of further work: the formalisation of safety requirements as predicates does not allow a formal verification of the consistency of the safety requirements (e.g., contradictory safety requirements can be formalised); the verification technique based on theory interpretation allows the creation of mappings that are syntactically correct but semantically wrong (e.g., visible display elements of the reference model can be mapped into state variables of the concrete model that are not rendered on the display); code generation is limited to Lisp code (new code generators that translate PVS models into C [20] and Java [11] are still under development). Further work is needed to demonstrate the approach for the entire user interface (we have illustrated the approach just for the data entry system). We have started to explore solutions to these limitations in [22] and [16].

Acknowledgements. This work is supported in part by the EPSRC (CHI+MED, EP/G059063/1), NSF CNS-1035715, and NSF CNS-1042829.

References

1. GPCA Hazards and Safety Requirements, <http://rtg.cis.upenn.edu/gip.php3>
2. The GPCA-UI Prototype, <http://tinyurl.com/QMUL-GPCA-UI>
3. Babamir, S.: Constructing a model-based software monitor for the insulin pump behavior. *Journal of Medical Systems* 36 (2012)
4. Cauchi, A., Gimblett, A., Thimbleby, H., Curzon, P., Masci, P.: Safer “5-key” number entry user interfaces using differential formal analysis. In: *BCS-HCI 2012* (2012)
5. Center for Devices and Radiological Health, U.S. Food and Drug Administration. White Paper: Infusion Pump Improvement Initiative (2010)
6. Harrison, M.D., Campos, J., Masci, P.: Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering* (2013)

7. Institute for Safe Medication Practices (ISMP). Guidelines for standard order sets, <http://www.ismp.org/tools/guidelines>
8. Jetley, R., Carlos, C., Purushothaman Iyer, S.: A case study on applying formal methods to medical devices. *International Journal on Software Tools for Technology Transfer* 5(4), 320–330 (2004)
9. Jetley, R., Jones, P.: Safety requirements based analysis of infusion pump software. In: *IEEE RTSS/SMDS* (2007)
10. Kim, B., Ayoub, A., Sokolsky, O., Lee, I., Jones, P., Zhang, Y., Jetley, R.: Safety-assured development of the GPCA infusion pump software. In: *ACM International Conference on Embedded software, EMSOFT 2011*. ACM (2011)
11. Lensink, L., Smetsers, S., van Eekelen, M.: Generating Verifiable Java Code from Verified PVS Specifications. In: Goodloe, A.E., Person, S. (eds.) *NFM 2012*. LNCS, vol. 7226, pp. 310–325. Springer, Heidelberg (2012)
12. Masci, P., Ayoub, A., Curzon, P., Harrison, M.D., Lee, I., Thimbleby, H.: Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In: *EICS 2013*. ACM Digital Library (2013)
13. Masci, P., Rukšėnas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., Thimbleby, H.: On formalising interactive number entry on infusion pumps. *ECEASST* 45 (2011)
14. Masci, P., Rukšėnas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., Thimbleby, H.: The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering* (2013)
15. Muñoz, C.: Rapid prototyping in PVS. Technical Report NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace (2003)
16. Oladimeji, P., Masci, P., Curzon, P., Thimbleby, H.: PVSio-web: a tool for rapid prototyping device user interfaces in PVS. To appear in *FMIS 2013* (2013)
17. Oladimeji, P., Thimbleby, H., Cox, A.: Number entry interfaces and their effects on error detection. In: Campos, P., Graham, N., Jorge, J., Nunes, N., Palanque, P., Winckler, M. (eds.) *INTERACT 2011, Part IV*. LNCS, vol. 6949, pp. 178–185. Springer, Heidelberg (2011)
18. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.K.: PVS: Combining Specification, Proof Checking, and Model Checking. In: Alur, R., Henzinger, T.A. (eds.) *CAV 1996*. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
19. Owre, S., Shankar, N.: Theory Interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Lab, SRI International, Menlo Park, CA (2001)
20. Owre, S., Shankar, N.: A brief overview of PVS. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) *TPHOLs 2008*. LNCS, vol. 5170, pp. 22–27. Springer, Heidelberg (2008)
21. Ray, A., Jetley, R., Jones, P., Zhang, Y.: Model-based engineering for medical-device software. *Biomedical Instrumentation & Technology* 44(6), 507–518 (2010)
22. Rukšėnas, R., Masci, P., Harrison, M.D., Curzon, P.: Developing and verifying user interface requirements for infusion pumps: a refinement approach. To appear in *FMIS 2013* (2013)
23. Shankar, N.: Efficiently Executing PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park (1999)
24. Shankar, N., Owre, S.: Principles and pragmatics of subtyping in PVS. In: Bert, D., Choppy, C., Mosses, P.D. (eds.) *WADT 1999*. LNCS, vol. 1827, pp. 37–52. Springer, Heidelberg (2000)

25. Thimbleby, H., Cairns, H.: Reducing number entry errors: Solving a widespread, serious problem. *Journal Royal Society Interface* 7(51) (2010)
26. UK National Patient Safety Agency. Design for patient safety: A guide to the design of electronic infusion devices (2010)
27. Xu, H., Maibaum, T.: An Event-B Approach to Timing Issues Applied to the Generic Insulin Infusion Pump. In: Liu, Z., Wasssyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 160–176. Springer, Heidelberg (2012)

Characterization of Failure Effects on AADL Models

Bernhard Ern¹, Viet Yen Nguyen², and Thomas Noll²

¹ Next Level Integration GmbH, Cologne, Germany
bern@next-level-integration.com

² RWTH Aachen University, Germany
{nguyen,noll}@cs.rwth-aachen.de

Abstract. Prior works on model-based Failure Modes and Effects Analysis (FMEA) automatically generate a FMEA table given the system model, a set of failure modes, and a set of possible effects. The last requirement is critical as bias may occur: since the considered failure effects are restricted to the anticipated ones, unexpected effects - the most interesting ones - are disregarded in the FMEA.

In this paper, we propose and investigate formal concepts that aim to overcome this bias. They support the construction of FMEA tables solely based on the system model and the failure modes, i.e., without requiring the set of effects as input. More concretely, given a system specification in the Architecture Analysis and Design Language (AADL), we show how to derive relations that characterize the effects of failures based on the state transition system of that specification. We also demonstrate the benefits and limitations of these concepts on a satellite case study.

1 Introduction

Safety and dependability assessments are imperative for the engineering of safety-critical systems. In particular, a clear understanding of how failures emerge, how they propagate and how these are dealt with is key towards trustworthy operation of the system itself. Methods such as Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) contribute to this. These two techniques relate faults to failures and failures to effects. In current engineering practice, these analyses are conducted manually. With the advent of model-based engineering, the trend is towards more automated techniques. In [2], an approach is used that generates an FMEA table from a system model expressed in the Architecture and Analysis Design Language (AADL). The concepts described in this paper improve upon it by reducing bias and allow for a more explorative analysis of failures and their effects.

This paper is organized as follows. Section 2 introduces our modeling language, a dialect of AADL. Then Section 3 describes and explains *effect relations* and *effect matrices*, which are our primary contributions. In Section 4, our prototypical tool is explained and our concepts are validated on a satellite case study based on [6]. Sections 5, 6 and 7 respectively deal with related work, future work, and conclusions.

2 Preliminaries

The Architecture Analysis and Design Language (AADL) is an industry standard for modeling safety-critical system architectures, and is designed and governed by the Society of Automotive Engineers (SAE) [1]. The AADL dialect that we are using in this paper, just referred to as AADL in the following, has been developed within a project entitled Correctness, Modeling and Performance of Aerospace Systems (COMPASS) that was funded by the European Space Agency (ESA) [3]. It provides a cohesive and uniform approach to model heterogeneous systems, consisting of software and hardware components, and their interactions. Furthermore, it has been drafted and enhanced with the following essential features in mind:

- Modeling both the system’s nominal and faulty behavior. To this aim, primitives are provided to describe software and hardware faults, error propagation (that is, turning fault occurrences into failure events), sporadic (transient) and persistent faults, and degraded modes of operation (by mapping failures from architectural to service level).
- Modeling (partial) observability and the associated observability requirements. These notions are essential to deal with diagnosability and Fault Detection, Isolation and Recovery (FDIR) analyses.
- Specifying timed and hybrid behavior. In particular, in order to analyze physical systems with non-discrete behavior, such as mechanics and hydraulics, the modeling language supports continuous real-valued variables with (linear) time-dependent dynamics.
- Modeling probabilistic aspects, such as random faults, repairs, and stochastic timing.

A complete AADL specification consists of three parts, namely a description of the nominal behavior, a description of the error behavior and a fault injection specification that describes how the error behavior influences the nominal behavior. These three parts are discussed in order below.

Nominal Behavior. The system model is hierarchically organized into components, distinguished into software (processes, threads, data), hardware (processors, memories, devices, buses), and composite components (called systems). Components are defined by their type and their implementations.

The component type specifies the ports through which the component communicates with its environment. There are two kinds of ports, namely event and data ports. Event ports enable components to synchronize their state upon each other whereas data ports are used to expose component variables to neighboring components.

A component implementation (such as the one given in Listing 1) describes the internal structure of a component through the definition of its subcomponents (lines 2–3), their interaction through (event and data) port connections, the (physical) bindings at runtime and the behavior via modes (lines 4–6) and

transitions (lines 7–10) between them. This behavior, which basically is a finite state automaton, describes how the component evolves from mode to mode while being triggered by events, or by spontaneously triggering events at the ports. Upon a transition, data components (like integer, real and Boolean variables) may change values due to transition assignments. Modes can be further annotated with invariants (e.g. the expressions after **while** on lines 5–6) on the value of data components (continuous or clock variables), restricting for example residence time. They furthermore may contain trajectory equations, specifying how continuous variables evolve while residing in a mode. (Here **energy'** refers to the first derivative of the energy value.) This is akin to timed and hybrid automata. Mode transitions may give rise to modifications of a component's configuration: subcomponents can become (de-)activated and port connections can be (de-)established. This depends on the “in modes” clause, which can be declared along with port connections and subcomponents.

Listing 1. An example battery component implementation.

```

1 system implementation Battery.Imp
2   subcomponents
3     energy: data continuous default 1.0;
4   modes
5     charged: activation mode while energy' = -0.02 and energy >= 0.2;
6     depleted: mode while energy' = -0.03 and energy >= 0.0;
7   transitions
8     charged -[then voltage := 2.0 * energy + 4.0]-> charged;
9     charged -[empty when energy = 0.2]-> depleted;
10    depleted -[then voltage := 2.0 * energy + 4.0]-> depleted;
11 end Battery.Imp;

```

Error Behavior. Error models are an extension to the specification of nominal models and are used to conduct safety, dependability and performability analyses. For modularity, they are defined separately from nominal specifications. Akin to nominal models, an error model is defined by its type and its associated implementations.

An error model type defines an interface in terms of error states and (incoming and outgoing) error propagations. Error states are employed to represent the current configuration of the component with respect to the occurrence of errors. Error propagations are used to exchange error information between components.

An error model implementation (such as the one given in Listing 2) provides the structural details of the error model. It defines a (probabilistic) machine over the error states declared in the error model type. Transitions between states (lines 6–10) can be triggered by error events (lines 2–5), reset events, and error propagations. Error events are internal to the component; they reflect changes of the error state caused by local faults and repair operations, and they can be annotated with occurrence distributions to express probabilistic error behavior. Moreover, reset events can be sent from the nominal model to the error model of the same component, trying to repair a fault that has occurred. Whether or

not such a reset operation is successful has to be modeled in the error implementation by defining (or respectively omitting) corresponding state transitions. Outgoing error propagations report an error state to other components. If their error states are affected, the other components will have a corresponding incoming propagation.

Listing 2. An example battery error model implementation.

```

1 error model implementation BatteryFailure.Imp
2   events
3     die: error event occurrence poisson 0.001;
4     works: error event occurrence poisson 0.2;
5     fails: error event occurrence poisson 0.8;
6   transitions
7     ok -[die]-> dead;
8     dead -[reset]-> resetting;
9     resetting -[works]-> ok;
10    resetting -[fails]-> dead;
11 end BatteryFailure.Imp;

```

Fault Injections. As error models bear no relation with nominal models, an error model does not influence the nominal model unless they are linked through fault injections. They describe the effect of the occurrence of an error on the nominal behavior of the system. More concretely, a fault injection specifies the value update that a data element of a component implementation undergoes when its associated error model enters a specific error state.

Multiple fault injections between error models and nominal models are possible. An automatic procedure, the so-called model extension, is employed to integrate both models and the given fault injections. It yields a combined specification that represents both the nominal and the faulty behavior of the system. Its semantics is formally defined by a transition system (cf. Definition 1) whose states are determined by the current modes and error states of all components, together with the current values of their data elements. Its transitions are derived from both the (nominal) mode and the (faulty) error state transitions, attaching a unique *transition label* to each. The latter can be used by the subsequent analyses, such as the failure effect analysis which is described in the present paper, to distinguish different types of transitions. More details on the specification language and its formal semantics can be found in [2].

3 Characterizing Effects

The two main contributions of this paper are described in this section, namely *effect relations* and *effect matrices*. A simple algorithm shall be sketched for computing these.

3.1 Effect Relations

We determine effect relations over a formal model of an AADL specification called a *transition system*, which captures all its possible behaviors. It is defined as follows:

Definition 1 (Transition System). *A transition system is a tuple $\langle S, T, \rightarrow, I \rangle$ where*

- S is a set of states,
- T is a set of transition labels,
- $\rightarrow \subseteq S \times T \times S$ is the transition relation, and
- $I \in S$ is the initial state.

Here the set T contains AADL transition labels. These are the transitions occurring *syntactically* in the AADL model. For example, line 8 in Listing 1 yields one transition label, as well as lines 9 and 10 in the same example. Also transitions in the error model are part of T , e.g. each of lines 7–10 in Listing 2.

Notationwise, we use several shorthands. A transition $\langle s, t, s' \rangle \in \rightarrow$ is also noted as $s \xrightarrow{t} s'$. It means that the AADL transition (labelled as $t \in T$) was applied to s to reach state s' . Also, we use $t(s)$ which, given $t \in T$ and $s \in S$, is defined as $t(s) = s'$ if $s \xrightarrow{t} s'$. Otherwise $t(s) = \perp$, i.e. undefined, where that $t(\perp) = \perp$. This shorthand assumes that the transition system is deterministic with respect to the set of labels T . This assumption holds for AADL, since non-deterministic behavior cannot be expressed in a *single* AADL transition. Furthermore, we describe the set of active transitions $A(s)$ as $\{t \in T \mid \exists s' : s \xrightarrow{t} s'\}$ (assuming that $A(\perp) = \emptyset$). Now we can define a binary effect relation over transitions in T :

Definition 2 (Effect Relation). *Given $s \in S$ and $t_1, t_2 \in T$, t_1 and t_2 can be in relation Independent (\parallel), Dependent (\bowtie), Conflict ($\#$) or Enable (\triangleleft). These relations are defined as*

- $t_1 \parallel t_2$ if $t_1(t_2(s)) = t_2(t_1(s))$ and $t_1(t_2(s)) \neq \perp$.
- $t_1 \bowtie t_2$ if $t_1(t_2(s)) \neq t_2(t_1(s))$ and $t_1(t_2(s)) \neq \perp$ and $t_2(t_1(s)) \neq \perp$.
- $t_1 \# t_2$ if $t_1, t_2 \in A(s)$ and $t_2 \notin A(t_1(s))$.
- $t_1 \triangleleft t_2$ if $t_2 \notin A(s)$ and $t_2 \in A(t_1(s))$.

Otherwise, the transitions are unrelated in s . For $\bowtie \in \{\parallel, \bowtie, \#, \triangleleft\}$, the notation $s \models t_1 \bowtie t_2$ means that in state s , the transitions t_1 and t_2 are in relation \bowtie .

The effect relations are depicted in Figure 1. Intuitively, the independency relation indicates that the transitions do not affect each other. Observe that this relation is symmetric, i.e. $t_1 \parallel t_2$ if and only if $t_2 \parallel t_1$. The dependency relation means that the related transitions affect each other, i.e. that changing the order in which they occur results in different states. Also this relation is symmetric. The conflict relation means that one transition disables another. Thus when $t_1 \# t_2$, transition t_2 is disabled and thus inactive after execution of transition t_1 . The enable relation is of most interest to our application, namely characterizing

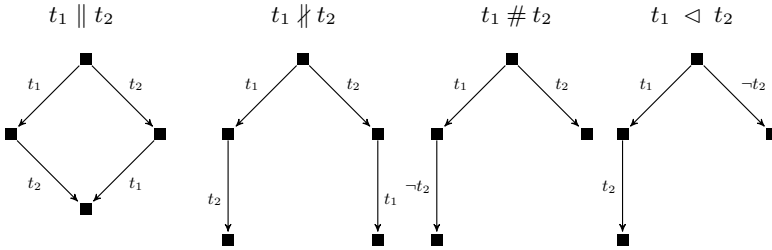


Fig. 1. The four effect relations $\parallel, \not\parallel, \#$ and $<$

failure effects. This relation intuitively expresses that one transition can cause another to happen. Hence when $t_1 < t_2$, it means that t_2 is caused by t_1 . Both the conflict and the enable relations are generally non-symmetric.

Until now, the effect relation is defined for unsynchronized transitions. In our AADL dialect, transitions can however synchronize. The above definitions can easily be lifted to support this by generalizing the definition of a transition relation to the type $\rightarrow \subseteq S \times T \times (T \cup \{\epsilon\}) \times S$. Here $s \xrightarrow{t, \bar{t}} s'$ means that \bar{t} is the rendez-vous transition interacting with t . In case t is unsynchronized, $\bar{t} = \epsilon$. The definition for effect relations are then lifted similarly as well. Due to page limit constraints, we refer the reader to [5] for the lifted definition. In the remainder of this paper we concentrate on unsynchronized transitions, since this keeps the notations simpler. We will bear in mind that all the techniques described in the following can easily support synchronized transitions by lifting definitions.

Furthermore, it is important that no transitions are left unnoticed in the effect relation, meaning that in any state s , any pair of transitions $t_1 \in A(s)$ and $t_2 \in A(s) \cup A(t_1(s))$ are related by exactly one of the independency, dependency, conflict or enable relations. The proof of these results is given in [5], and the interested reader is referred to this publication due to lack of space.

3.2 Effect Matrix

Given the definition for effect relations, we now determine how one transition affects another. This can be done by traversing the transition system and, in each state, determining which effect relation applies to all pairs of active transitions. The results are then accumulated in a matrix which we call the effect matrix.

Definition 3 (Effect Matrix). *Given a transition system $\langle S, T, \rightarrow, I \rangle$ with $T = \{t_1, \dots, t_n\}$, the effect matrix M^\bowtie for relation $\bowtie \in \{\parallel, \not\parallel, \#, <\}$ is given by:*

$$M^\bowtie := (m_{ij}^\bowtie)_{1 \leq i, j \leq n} \quad \text{where} \quad m_{ij}^\bowtie = \{s \in S \mid t_i \neq t_j \wedge s \models t_i \bowtie t_j\}$$

A simple procedure to compute M^\bowtie given a transition system is shown in Algorithm 1. The time complexity for constructing M^\bowtie is polynomial in the number of states $|S|$ and transitions $|T|$. This (rather naive) algorithm traverses

Algorithm 1. Compute effect matrix M^{\bowtie} .

Input: $\langle S, T, \rightarrow, I \rangle$ **Output:** M^{\bowtie}

```

1:  $m_{ij} := \emptyset \quad \forall t_i, t_j \in T$ 
2: for all  $s \in S$  do
3:   for all  $t_i \in A(s)$  do
4:     for all  $t_j \in A(s) \cup A(t_i(s))$  do
5:       if  $s \models t_i \bowtie t_j$  then
6:          $m_{ij}^{\bowtie} := m_{ij}^{\bowtie} \cup \{s\}$ 

```

all states in S (line 2) and then determines for all pairs of active transitions in two transition steps (lines 3 and 4) how they are related (line 5). This is the algorithm we implemented. In the worst case, all transitions in T are always active, resulting in a time complexity of $O(|S| \times |T|^2)$. However, in practice only a small fraction of the transitions T are active, making the size of the state space $|S|$ the most important factor. The set of active transitions $A(s)$ is pre-computed during state space generation, and hence does not add additional costs.

The effect matrix is key to estimating effects, and failure effects in particular. Especially the enable relation (\triangleleft) is of importance for that. If a pair of transitions *only* occurs as $t_1 \triangleleft t_2$ in the state space, then t_1 and t_2 can be considered as *causally* related. Typically, t_1 would be an error transition originating in the AADL error model and t_2 would be a change to the nominal data ports or subcomponents as consequence to the error. This is demonstrated in the next section, where we report on our case study.

4 Experimental Evaluation

In the following, we detail on our implementation of the concepts introduced in Section 3 and demonstrate their usefulness on an adaptation of the satellite case study described in [6].

4.1 Tool

We developed a prototypical tool in Java that takes an AADL model as input, and then computes the effect matrix (see Definition 3 and Algorithm 1) as result. Inside the tool, it uses our AADL-to-Promela translator [10] to have the SPIN model checker [9] generate the transitions system. This transition system, in explicit-state representation, is dumped to disk as a file. All computations are then performed on that state space.

4.2 Case Study

We computed effect matrices for adaptations of a satellite case study as described in [6]. The adaptation reduces the original case to only cover the

Attitude & Orbit Control System (AOCS) and its fault management system, making its size (16008 states) more amenable to validating our approach. The AOCS is a control system responsible for maintaining the satellite's orientation. It is equipped with sensors (e.g. gyroscopes, Earth trackers, Sun trackers, magnetometers) to determine its current orientation. It uses actuators (e.g. reaction wheels, magnetic torquers, thrusters) to modify its orientation if required. These components may however degrade and/or fail due to space hazards, in particular radiation. So during the design of the AOCS, fault-tolerant aspects need to be incorporated. For example, components are redundantly equipped and failure compensation techniques are integrated. In our case study, we are in particular interested in the failure detection, isolation and recovery (FDIR) procedures. These procedures continuously monitor the system for non-nominal phenomena. If one is detected, FDIR attempts to determine its cause and subsequently initiates a corresponding (recovery) procedure. A minimal downtime is desired while doing so.

For demonstrating and validating our approach, we focus on one of the recovery procedures, namely the one that handles Earth tracking sensor failures. These sensors may for example fail to provide signals to the remainder of the AOCS. If this situation is left unhandled, it may cause the system to attain an incorrect orientation. The involved AOCS subsystems are

- Primary Earth Sensor (ES_A)
- Secondary Earth Sensor (ES_B)
- Control and Data Unit (CDU), containing
 - Processor Module (PM)
 - On-Board Data Handling software (OBDH), containing
 - * Earth Sensors Control Software (ES_CTRL)
 - * Earth Sensors Failure Detection, Isolation and Recovery Software (ES_FDIR)

There are many more components present, but we omit them here. Furthermore, the satellite may reside in particular modes of operation. In normal conditions, it is in the Nominal Mode. Upon failures, it is expected to switch to the Degraded Mode. If the recovery is successful, the satellite is expected to return to Nominal Mode. If not, it should switch to Safe Mode, in which ground control takes further action. There is also an Orbit Control Mode during which the satellite performs trajectory corrections. While switching to different modes, the topology of the system is reconfigured by disabling/enabling components and rerouting data and command streams from disabled to enabled components.

A particular recovery procedure consists of the following sequence of transitions (indicated by t_i), starting with error transition t_1 that causes the failure:

- t_1 : Injection of signal loss error in primary Earth sensor.
- t_2 : ES_FDIR detects signal loss.
- t_3 : ES_FDIR changes AOCS mode from Nominal to Degraded.
- t_4 : ES_FDIR isolates the failure and initiates switch-over of Earth sensors.
- t_5 : ES_CTRL disables primary Earth sensor.

t_6 : ES_A sets its status flag to off.

t_7 : ES_CTRL enables secondary Earth sensor.

t_8 : ES_B checks its power and sets its status flag to on.

t_8 : ES_FDIR changes AOCS mode from Degraded to Nominal.

This recovery procedure triggers transitions in manifold components (e.g. ES_FDIR, ES_CTRL, ES_A, ES_B), which are part of the AOCS, CDU, PM and OBDH subsystems. The satellite model captures all these interrelations. The Earth sensor FDIR component furthermore includes many more behaviors (and thus more transitions) to cover other scenarios, like the transition to Safe Mode after a signal loss of the secondary Earth sensor while being in Orbit Control Mode. Another example is the switch-over that is initiated when the primary Earth sensor has failed while being in that mode. This makes the FDIR component tightly coupled with a major part of the overall system. It is therefore imperative to understand which effects the FDIR component has on the system, and under which (isolated) conditions these effects apply. This aims to avoid undesired behaviors.

When we computed the effect matrix for a scenario in which the primary Earth sensor fails, we were able to determine the exact recovery procedure chain, namely that $t_1 \triangleleft t_2 \triangleleft \dots \triangleleft t_8$. The effect matrix also showed that this order is strict, i.e., there are no cases where those transitions were executed in another order (e.g. $t_7 \triangleleft t_5$). This proves that the recovery procedure as implemented in the model is indeed restricted to the occurrence of the failure, namely transition t_1 . The effect matrix can thus be distilled to a FMEA table entry where upon the occurrence of a primary Earth sensor signal loss, the effect is the chain of transitions from t_2 up to t_8 . More entries can be distilled by computing effect matrices for the models with different fault configurations, like a secondary Earth sensor failure.

In the first runs on this case study, we loaded up models in which the primary Earth sensor failed while the system was in Nominal Mode. What we then expected to see in the effect matrix were the effects $t_1 \triangleleft \dots \triangleleft t_8$. The effect matrix however also exhibited $t_4 \triangleleft t_{ocm}$ and $t_4 \triangleleft t_{eam}$, which involve the Telemetry Tracking & Control (TT&C) component that is used for communication with the satellite:

t_{ocm} : TT&C changes AOCS mode from Orbit Control to Nominal.

t_{eam} : TT&C changes AOCS mode from Earth Acquisition to Nominal.

The effects $t_4 \triangleleft t_{ocm}$ and $t_4 \triangleleft t_{eam}$ were unexpected to us because t_4 should only be occurring while being in Degraded Mode (due to transition t_3). Somehow, there was an interleaving in between that caused a transition to the Orbit Control Mode or to the Earth Acquisition Mode. The effect matrices furthermore showed that other transitions in the TT&C were enabled by other unexpected transitions. This indicated that our model did not handle the mutual exclusion of the TT&C with other components correctly. This was corrected in our final model.

4.3 Scalability

To obtain an impression of the scalability of our prototypical tool, we made a few adaptations of the satellite case study entailing varying sizes of the state space. The results are as follows:

Number of states	Generate state space [min]	Compute M^{∞} [sec]
10792	46	6
21584	86	14
31044	206	30

The column “Generate state space” gives the time needed for generating the state space by SPIN. The column “Compute M^{∞} ” lists the time needed for computing the effect matrix once the state space is generated. As we expected, the majority of time is necessary for generating the state space itself (which happens within an order of hours). Once this is accomplished, the state space is loaded into memory and is traversed in linear time. As shown in the table, this proceeds in an order of seconds. We also experimented with models beyond 31044 states, but for those our machine (2.1 GHz processor, 192 GB of RAM) runs out of memory during state space generation.

5 Related Work

Our concepts relate closely to approaches towards existing FMEA table generation techniques [2, 8]. With respect to [2], our approach overcomes the need to specify a set of *expected* failure effects as an input, and instead the effect matrix can be used to *discover* failure effects. This was the motivation of this work. With respect to [8], our approach uses the variant of AADL described in [2], which contrary to [8] incorporates behavioral aspects beyond the original AADL and its Error Model Annex. Furthermore, their approach appears (as details are scarce) to extract FMEA table entries from state space traces, and post-processes those entries using user-defined filters. Our effect matrix considers effect relations across multiple traces that globally hold in the state space, and hence is more fine-grained. Our concepts are not alternative to [2, 8], but rather complementary because of their ability to study finer effect relations.

Our approach borrows and incorporates concepts from existing works. The (in-)dependency relations are inspired by Mazurkiewicz trace equivalences [4], which capture both aspects. The conflict relation is inspired by the theory of event structures [11]. The enable relation is inspired by dynamic partial order reduction [7]. These concepts on their own are thus not novel, but their combination is, in particular when used for characterizing failure effects using an effect matrix.

6 Future Work: Entangled Effects

During our experimental evaluation, we observed failure effects whose faults are intertwined with other (not necessarily failure) effects. We call them *entangled*

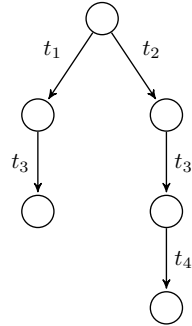
effects. We developed experimental approaches whose refinement and applicability require further investigation.

We observed that in the effect matrix, two transitions are not strictly in one relation throughout the state space. For example, it could occur that there exists a state s_k where $s_k \models t_i \triangleleft t_j$ while in another state s_l it holds that $s_l \models t_i \# t_j$. Thus, in one state one transition enables the other while in another state they conflict. With regard to the effect matrix, this means that both m_{ij}^{\triangleleft} and $m_{ij}^{\#}$ are non-empty. These deviating effects can be understood by comparing states s_k and s_l . We believe that constructing two paths, one leading to state s_l and the other to s_k , and then comparing them helps to understand the deviating effects. The two paths can be found by a backwards synchronized breadth-first search from s_k and s_l that proceeds until a common ancestor state is identified. Both paths, say $\pi_k = s_0, \dots, s_k$ and $\pi_l = s_0, \dots, s_l$, can then be sliced to emphasize the differences between the visited states. This may help for long paths or for states that are determined by many variables. The differences are computed by checking the valuations of variables between states.

Entanglement can also originate from effects caused by a combination of faults. Such a notion is useful for enhancing single-fault FMEA with multiple-failure configurations. In case of independent multiple failures, our concepts captures those as independent transitions which all enable a common effect.

A similar conceptualisation for multiple dependent failures is more involved. For this, we need to relate transitions by transitivity. In this paper, our definition of effect relations only considers pairs of transitions that directly follow each other in the state space. This led us to distill the effect relation of $t_1 \triangleleft \dots \triangleleft t_8$ in our case study. However, this logically means by transitivity that $t_1 \triangleleft t_8$. The latter relation is however not detectable by our current definition.

We have been experimenting with transitive effect relation definitions in [5] to overcome this. However, we have not yet developed a suitable definition in which effect relations do not become over-approximated. For example, a transitive rule stating that $t_1 \triangleleft t_3 \wedge t_3 \triangleleft t_4$ implies that $t_1 \triangleleft t_4$ does not generally hold. The figure on the right exemplifies this situation. It shows that transitions t_1 and t_4 are in conflict rather than enabling each other. Hence we regard this topic as future work.



7 Conclusion

In this paper, we propose an approach for characterizing failure effects on AADL models by automatically detecting effect relations over the state space underlying the AADL model. This is of particular value to the automated generation of precise FMEA tables, although not being limited to this application. Our approach is also not limited to AADL models, but is of a more general nature. An effect matrix can be computed for any model equipped with a formal semantics that describes its behavior by means of a labeled transition system. Furthermore,

the effect matrix is not limited to characterizing failure effects. It also captures nominal effects, yet we believe it is of most applicable value to the safety and dependability domain. This is also demonstrated by our evaluation using a satellite case study.

Our work also revealed directions for future work. In particular, entangled effects through transitivity are of immediate practical and theoretical interest. The overall outcome so far demonstrates and validates our concepts that aid in constructing precise fault-failure effect relations (e.g. FMEA tables) as rigorously and automatically as possible.

Acknowledgement. This work was partially supported by ESA/ESTEC (contract no. 4000100798) and Thales Alenia Space (contract no. 1520014509/01).

References

- [1] Architecture, Analysis and Design Language AS5506. SAE (2004)
- [2] Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, Dependability and Performance Analysis of Extended AADL Models. *Computer Journal* 54(5), 754–775 (2011)
- [3] COMPASS Project, <http://compass.informatik.rwth-aachen.de>
- [4] Diekert, V., Rozenberg, G. (eds.): *The Book of Traces*. World Scientific, Singapore (1995)
- [5] Ern, B.: *Model-Based Criticality Analysis by Impact Isolation*. Master's thesis. RWTH Aachen University (2012)
- [6] Esteve, M.-A., Katoen, J.-P., Nguyen, V.Y., Postma, B., Yushtein, Y.: Formal Correctness, Safety, Dependability and Performance Analysis of a Satellite. In: *Proc. 34th International Conference on Software Engineering (ICSE 2012)*, pp. 1022–1031 (2012)
- [7] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: *Proc. 32nd Symposium on Principles of Programming Languages (POPL 2005)*, pp. 110–121 (2005)
- [8] Hecht, M., Lam, A., Vogl, C., Dimpfl, C.: *A Tool Set for Generation of Failure Modes and Effects Analyses from AADL Models*. Presentation at Systems and Software Technology Conference 2012 (2012)
- [9] Holzmann, G.J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional (2003)
- [10] Odenbrett, M.R.: *Explicit-State Model Checking of an Architectural Design Language using SPIN*. Diplomarbeit. RWTH Aachen University (2010)
- [11] Winskel, G.: Events, Causality and Symmetry. *Computer Journal* 54(1), 42–57 (2011)

Derived Hazard Analysis Method for Critical Infrastructures

Thomas Gruber¹, Georg Neubauer¹, Andreas Weinfurter¹,
Petr Böhm¹, and Kurt Lamedschwandner²

¹AIT Austrian Institute of Technology GmbH
{thomas.gruber, georg.neubauer, andreas.weinfurter,
petr.boehm}@ait.ac.at

²Seibersdorf Laboratories GmbH
kurt.lamedschwandner@seibersdorf-laboratories.at

Abstract. Within the Austrian national security research programme KIRAS, a study on security against electromagnetic threats was conducted. Apart from a survey on existing literature about respective events and analyses on existing threats and possible protection measures, a novel risk analysis method was developed, based on a qualitative FMEA (Failure modes and effects analysis). The traditional FMEA sheet was split into several tables taking advantage from the limited set of electromagnetic interference causes and a general set of high-level consequences. The resulting tables of risk priority numbers allowed a good overview on which defence and which protection measures should be prioritized. Finally, the method was validated based on three scenarios of road vehicle convoys with respect to its applicability. This paper describes the approach developed for the modified FMEA method and its application to three vehicle convoy scenarios, discussing the value of the method and interpreting the results of the validation.

Keywords: Hazard analysis, FMEA, HAZOP, Critical Infrastructures, Intentional Electromagnetic Interference, Risk Priority Number.

1 Introduction

In recent years, infrastructures like energy supply or communication networks have attracted the attention of government authorities as their continuous and correct functioning is considered increasingly important. Infrastructures whose functioning is essential for the population and the institutions of the state are called critical infrastructures (CI); examples are energy grids, communication networks, governmental institutions or healthcare. National as well as European programs for critical infrastructure protection have been developed, cf. [1] and [2], which aim to raise awareness among the persons responsible, to set up CI risk management, and to support CI providers in establishing protective and reactive measures. In this context, the Austrian Federal Ministry for Transport, Innovation and Technology (bmvit) has

established a dedicated research program for CI protection in Austria, called KIRAS¹, by which also the research presented in this publication has been funded.

The reasons for CI failure can be twofold: Unintentional failure can be caused by wear-out of system parts, aging of components or human error. On the other hand intentional malfunctions through criminal or terroristic acts must also be considered. Attacks against CI can be committed with physical violence, for example as bomb attacks or as sabotage. But in recent years, attacks by electromagnetic interference with various frequency spectra and energy densities have become a non-negligible threat. Cyber-criminality is one way of compromising CI security by highly-sophisticated abuse of electromagnetic means. Disturbing radio transmissions by jamming is another. Moreover, irradiation of electronic systems with electromagnetic energy can cause malfunctions: Temporary failure by forcing an emergency re-boot of computer-controlled systems or even a permanent outage through thermal or voltage overload of sensitive electronic systems.

The project SEMB² analysed the threat potential which IEMI (intentional electromagnetic interference) poses to CI in Austria. The non-classified study analysed freely available literature on respective events, available IEMI weapons and their cost, and the state of the art with respect to technical protective measures. In addition, an appropriate risk analysis method was developed, which was intended to support CI providers in judging where in the infrastructure to prioritize improvements of protective measures. This paper describes the selection and adaptation of the risk analysis method, explains the effort-saving way of data capturing and shows how the risk assessment results are presented. Finally, some of the results are shown, which were obtained in the verification of the method on the basis of three vehicle convoy scenarios. The conclusions comprise a judgment on the benefit of the method and an outlook on further research planned.

2 Intentional Electro-Magnetic Interference IEMI

One of the most frequently used terms for electromagnetic threats is IEMI (Intentional Electro-Magnetic Interference). Wik und Radasky [3] give the following definition: “Intentional, malicious generation of electromagnetic energy, leading to coupling of noise or signals into electrical and electronic systems. This causes interception, disturbance or damage of such systems for criminal or terrorist purposes.”

IEMI devices are also used in legal contexts, namely by security and military forces, for instance for stopping non-cooperative vehicles or for jamming communication with radio-controlled bomb triggers in order to ensure security for a vehicle convoy passing by.

The project SEMB treated most kinds of known IEMI devices with the only exception of nuclear bombs (for the NEMP = nuclear electromagnetic pulse), whose effects

¹ KIRAS is an artificial word combined from Greek *kirkos* (cycle) and *asphaleia* (security).

² The work described in this paper was funded by the Austrian Federal Ministry for Transport, Innovation and Technology (bmvit) within the Austrian national security research programme KIRAS (www.kiras.at).

were considered disastrous to such an extent that particular protection measures for single critical infrastructures seemed unrewarding. The following list enumerates the treated IEMI sources:

- Jammers
- Tasers (electroshock gun)
- ESD gun (electron-stream drilling gun)
- HPM (high-power microwave)
- UWB (ultra wide band)
- LEMP (lightning electromagnetic pulse)
- Security attack against communication and systems

Accidental, unintentional electromagnetic influences were normally not in scope with the only exception of geomagnetic storms. And the LEMP (lightning electromagnetic pulse) was considered as analogon for the effects of the HEMP.

With few exceptions, most IEMI weapons are operated in the frequency range between 25 MHz and 6 GHz, corresponding to free-space wavelengths between 12 m and 5 cm. This is due to the fact that – for coupling in - electromagnetic waves need slots or wires as antennas in the magnitude of a quarter wave length.

The probability that IEMI weapons are used is – among other aspects - associated with the availability of the devices, their cost, and the technical difficulty for constructing or manufacturing them. In the following, short descriptions of the mentioned classes of IEMI devices are given with indications how potential perpetrators can purchase or construct them.

Jammers.

Jammers are transmitters used to disturb radio communication by interfering the useful signal with a signal of higher power density at the same frequency. Devices have mostly multi-band or broadband characteristics and are available in different sizes (handheld, bag-sized or van-transportable boxes) and signal strengths (a few Watts up to Kilowatts with coverage from some tens of meters up to kilometres). They are legally available for military, security companies and authorities, but also easily purchasable for non-authorized users via internet from dubious sources.

Tasers.

Usually driven by a pressurized gas, the Taser fires two dart-like electrodes with barbed hooks at the target, which remain connected with the gun by isolated wires. Electroshock guns are normally used by police or security forces in order to paralyze a perpetrator with a short high voltage pulse at a peak current of typically several Amperes. Once the electrodes have hit the target, repeated application of electroshocks is possible. For unauthorized users, Tasers are available on the black market.

ESD guns.

ESD (electron-stream drilling) guns are commercially available tools which are normally used in EMC (electromagnetic compatibility) laboratories for testing electronic devices with respect to robustness against electrostatic discharges. In conformance

with different applicable standards, e.g. [4], ESD guns use a capacitor of 150pF or 330pF and a resistor of 330 Ω or 2k Ω for discharge of energy in the magnitude of some tens of mJ at peak voltages of up to 30 kV.

HPM.

HPM (high-power microwave) devices are often narrow-band generators operating at a fixed microwave frequency and are based on magnetrons; they can be constructed from RADAR units (pulsed magnetron) or by modifying a microwave oven (continuous wave). The radiation can be bundled very well by parabolic reflectors yielding very high energy densities at remarkable distances. Military versions in the size of vans can operate at distances of kilometres with pulse powers in the range of MW. HPM devices can destroy electronic devices; they are normally not available for unauthorized persons but can be built by modifying a commercially available microwave oven, provided some technical abilities.

Another narrowband source is the Marx generator, which uses a high voltage pulse coming from capacitors which are charged and switched in series to produce a damped sine signal in the range of MHz or GHz at peak power values of MW. Constructing a Marx generator requires advanced technical capabilities.

UWB.

An example for UWB (ultra wide band) devices is the EPFCG (Explosively Pumped Flux Compression Generator). It produces a high energy pulse with a short rise time using a high voltage peak created by a sudden change of the applied magnetic flux in a current-carrying coil; this is achieved by compressing the coil by an explosion. Such weapons are for one-shot use only but can be constructed relatively small in size and thus as portable devices.

3 Risk Evaluation

This section gives an introduction to metrics and analysis methods for risk evaluation and presents finally the decision for a risk analysis method for critical infrastructures.

3.1 The Term Risk

Risk is defined as the combination of the probability of occurrence of an unwanted event, and the damage caused by it; mathematically formulated it is the product of them (cf. [5]). The probability of occurrence is a dimensionless value between 0 (for “occurs never”) and 1 (indicating “occurs surely”). The extent of the damage is measured in the number of casualties or of fatalities or as the financial damage. Though it sounds cynical, the damage through fatalities is often given as the “value of statistical life”, cf. [7], i.e. how much society is willing to pay to prevent the loss of one human life, or, in other words, the marginal cost of avoiding one fatality. The value depends strongly on the gross domestic product of the country and varies extremely between developed and third world countries.

3.2 Risk Assessment

Two methods for determining the risk level in a risk assessment are commonly used in safety standards: Risk matrix and Risk Priority Number.

Risk Matrix.

The Risk matrix helps evaluate if a risk given by an estimated probability of occurrence (frequency) and a classification of the damage (consequence) is acceptable or not; Fig. 1 shows an example for a risk matrix.

Frequency	Consequence			
	Catastrophic	Critical	Marginal	Negligible
Frequent	I	I	I	II
Probable	I	I	II	III
Occasional	I	II	III	III
Remote	II	III	III	IV
Improbable	III	III	IV	IV
Incredible	IV	IV	IV	IV

Fig. 1. Risk matrix (Source: IEC 61508-5 ed.2)

Risk Priority Number RPN.

As can be seen in equation (1), the RPN is defined as the product of probability of occurrence O, severity S and “detection” D; the latter term may be misleading as it stands actually for the probability that the hazard or the faulty state remains undiscovered before it results in a failure.

$$RPN = O \cdot S \cdot D \tag{1}$$

O, D and S are estimated by a team of experts who rank the considered hazard with respect to the three factors of the RPN, each in a range between 1 and 10. It is evident, that the term “probability” used in this context is not equivalent to the mathematical value defined in probability calculus. It is in fact a qualitative rank where 1 stands for the least harmful value, 10 for the worst. As a result, the RPN is computed as a numeric value between 1 and 1,000. In this metric, “1” stands for an event with no hazardous consequences which is always detected in advance but never occurs. “1,000” means a catastrophic event which happens regularly or continuously and is never detected before the catastrophe occurs.

3.3 Selection of an Appropriate Risk Analysis Method

For analysis and assessment of the risks through electromagnetic threats, an appropriate risk analysis method had to be selected, which is generally applicable to different kinds of infrastructures and fits well for the specific requirements and conditions.

In the CI analysis we assumed to be faced with a high degree of uncertainty and complexity, and, apart from risk identification, we expected results wrt. consequences, probability of occurrence, level of risk, and the possibility to avoid the hazard.

In literature, many risk analysis methods can be found with varying level of detail and for different degree of knowledge about the system under consideration, for instance in [8]. ISO/IEC 31010:2009 [6] treats 31 risk analysis methods without making claims of being exhaustive. The standard contains short descriptions and provides guidance which technique is appropriate for obtaining certain result values in a particular setting with specific influencing factors (cf. tables A.1 and A.2 in [6]). Applying the above mentioned selection criteria to the tables in [6], we identified only five basically adequate methods: HAZOP [hazard and operability] study, SWIFT (Structured “What-if” technique), Scenario analysis, FMEA [Failure modes and effects analysis], and Cause-consequence analysis. We finally looked more closely at HAZOP and FMEA, two detailed methods which we were familiar with, and we had to choose between them.

The **HAZOP study** was originally developed for the chemical process industry. The analysis team applies a set of so-called guidewords (like “OTHER THAN” or “MORE”) to each combination of a process step (for instance a vessel) with a parameter (e.g. temperature) and analyses possible deviations from design or process intent expressed by the combinations with respect to causes, consequences, and existing detection as well as risk mitigation measures; finally, further risk reduction measures are recommended if necessary.

However, a reasonable interpretation can be found only for part of the combinations; the rest is meaningless and must be dropped. A short description of the HAZOP study can be found in [6] and [8]; there are also specific standards available like [9] describing the procedure in detail. It has to be mentioned that HAZOP studies are today used for a wide variety of systems including electrical, electronic and computer controlled systems.

FMEA was originally developed by the US department of defence for military purposes; today it is one of the first choices for performing a system safety analysis. It shall be mentioned that, apart from this System FMEA, there are also variants for process improvement called Process FMEA. According to the system structure, all components are analysed with respect to possible failures, and – similar to the HAZOP study - causes, consequences, and detection as well as risk mitigation measures are analysed and improvements proposed. Unlike the HAZOP, however, FMEA uses neither parameters nor guidewords but relies on the expertise and creativity of the analysis team to discover all potential failure modes of the system components.

Two basic variants of a system FMEA can be distinguished, depending on knowledge about failure probabilities:

- A quantitative FMEA, which calculates the risk for each single failure mode using a numeric failure probability value, which can be derived from own statistical data with comparable systems or components if such data is available, or, otherwise, from catalogues for failure prediction like for instance [10] or [11].
- A qualitative FMEA, which follows basically the same procedure as the quantitative variant but qualifies the probability of occurrence according to a qualitative

scheme ranging from “extremely improbably or never” until “extremely frequent or always”. For risk ranking, the above described risk matrix and the risk priority number can be used.

Both, HAZOP and FMEA, allow hazard analyses of systems at different levels of detail and are basically adequate for the purpose. Considerations about analysis effort and expected benefit of the technique, however, lead us to a conclusive decision.

The analysis of IEMI threats to hardware differs in several aspects from a classical safety analysis, which targets mainly at stochastic failures of system components due to aging and wear-out:

1. The set of (high-level) causes considered is well-known and limited to few kinds of intentional electromagnetic attacks.
2. Applicable electro-technical protective measures are widely independent of the individual infrastructure; they are rather associated with the types of electronic control and supervision equipment used.
3. In the absence of sufficient statistical data³, computations with failure rates in the sense of expected values as used in probability calculus are not applicable. Moreover, political situation, expected criminal energy and local accessibility play a role.

The first difference makes clear that the high coverage of potentially unknown deviations revealed in a HAZOP study brings no additional value, while the - according to our own experience - high number of meaningless combinations causes significant additional effort. The first two differences yield potential for an effort-saving approach in which causes and risk prevention are treated on a general level. The third difference indicates that - in the absence of statistical failure probabilities - a quantitative approach is not feasible.

Therefore a qualitative FMEA using RPN for risk ranking has been chosen, which was, in addition, modified, avoiding redundant data input for the above mentioned general data.

4 Adaptation of FMEA for CI

The main goal was to identify the most critical subsystems or components, for which protection measures should be prioritized. As indicator for this decision, the risk priority number was used.

Protection measures can be of technical nature like for instance overvoltage protection, but also organizational security measures are in scope. Observations of trends in the criminal and terrorist scene can deliver additional information on the expected kinds of threats. It is therefore of interest to look at the total risk for single components as well at the risk potential through certain kinds of electromagnetic threats. For obtaining this information, we needed RPN mean values for single IEMI threats as well as for single endangered components.

³ In the non-classified study SEMB, we had no access to classified material. Therefore, we do not know if useful statistical data is available at all.

Project:				Version:			Date:				
System:				Subsystem:			Teamwork leader:				
Id.	Comp onent	Function	Failure mode	Failure cause	Local effects	Global effects	S	O	D	RPN	Corrective actions

Fig. 2. Typical qualitative FMEA sheet (Source: M. Rausand, NTNU)

Our attempts to create an appropriate risk analysis method started from a qualitative FMEA, for which a typical work sheet is shown in Fig. 2.

It is evident that this table becomes confusingly large when it is filled with all IEMI induced failure modes, all IEMI causes, and all protective measures, and it becomes hard to identify trends for certain IEMI threats or the total risk potential for certain components. As mentioned in the above section, the risk analysis method was therefore adapted to the specific needs of IEMI threat analysis and with view to effort optimization.

One simplification we could make was the confinement to two kinds of global consequences (actually, our analysis focused on *global* consequences):

- Temporary failures, which disappear when the IEMI influence ends or when the system reboots, and
- Permanent failures due to damage of electronic components.

Both consequences were handled in separate sets of tables, the latter are described in the following.

Instead of the classical FMEA table we chose to collect data in separate tables, one for Occurrence, one for Detection and one for Severity. The Occurrence and the Detection tables had each one column per endangered component and one row for each IEMI threat. For Severity, the table had one row per endangered component, and columns for local and global effects of temporary and persistent failures as well as columns for reparability and replacement. From this data we computed tables with the RPN including mean values for each IEMI threat and for each endangered component. The following section on the validation of the method contains examples for each of the mentioned tables.

5 Validation of the Risk Analysis Method

As a final step, the derived hazard analysis method described above was validated for one kind of critical infrastructure, namely vehicle convoys.

5.1 Selected Scenarios

It was agreed among project partners that civil and military convoys should be in scope, so three different scenarios were selected:

- a VIP convoy for a state visit within Vienna without police road blocks,
- a convoy of fire trucks driving to a disaster operation (inundation),
- a convoy of military trucks driving to a field manoeuvre in Austria.

5.2 Data Capturing

As, according to our literature analysis, no statistically relevant amount of empirical data about IEMI events was available, risk had to be determined by estimation. For this purpose, an analysis team comprising all relevant stakeholders (including personnel of the CI operator) should evaluate the risks in analysis meetings. However, effort in the magnitude of person-weeks or even person-months would have to be spent for this purpose. For a first validation, this effort-consuming involvement of CI operators, who were themselves not project partners, should be avoided. For this reason, the estimation was performed by personnel of the research institute AIT only. For the concrete RPN tables obtained in the validation, we therefore have to state that the values are provisional and should be seen as examples only.

The estimated probabilities were based on assumptions on the risk for typical technical equipment in Austria. The following tables show examples of data we captured.

Table 1. Estimation of the probability of a temporary failure in a VIP convoy

<i>Temporary failure -VIP convoy</i>	<i>Endangered components</i>								
<i>IEMI threats</i>	<i>Safety-critical control unit</i>	<i>Non-safety-critical control unit</i>	<i>Built-in TETRA radio set</i>	<i>Mobile TETRA radio set</i>	<i>Mobile analogue radio set</i>	<i>Built-in electronics for special equipment not needed for mission</i>	<i>Built-in electronics for special equipment needed for mission</i>	<i>Permanently mounted emergency light</i>	<i>Mobile emergency light</i>
<i>Jamming</i>	2	2	4	4	4	1	1	1	1
<i>Taser</i>	1	2	2	1	1	1	1	1	1
<i>ESD gun</i>	1	1	2	1	1	1	1	1	1
<i>HPM</i>	3	4	5	4	4	1	1	2	2
<i>UWB</i>	3	4	5	4	4	1	1	2	2
<i>LEMP</i>	2	3	4	3	3	1	1	1	1
<i>Security attack</i>	1	1	5	5	1	1	1	1	1

Table 2. Assessment of detection values for a fire truck convoy

<i>Detection - fire truck convoy</i>	<i>Endangered components</i>								
<i>IEMI threats</i>	<i>Safety-critical control unit</i>	<i>Non-safety-critical control unit</i>	<i>Built-in TETRA radio set</i>	<i>Mobile TETRA radio set</i>	<i>Mobile analogue radio set</i>	<i>Built-in electronics for special equipment not needed in mission</i>	<i>Built-in electronics for special equipment needed for mission</i>	<i>Permanently mounted emergency light</i>	<i>Mobile emergency light</i>
<i>Jamming</i>	10	10	10	10	10	10	10	10	10
<i>Taser</i>	1	1	1	1	1	1	1	1	1
<i>ESD gun</i>	1	1	1	1	1	1	1	1	1
<i>HPM</i>	9	9	9	9	9	9	9	9	9
<i>UWB</i>	9	9	9	9	9	9	9	9	9
<i>LEMP</i>	1	1	1	1	1	1	1	1	1
<i>Security attack</i>	3	3	3	3	3	3	3	3	3

Note: Very high values for HPM, UWB and Jamming in **Table 2** point at the fact that – due to the long way - it is hardly possible to detect these threats in advance.

Table 3. Consequences of a failure in the fire trucks convoy

<i>Affected component</i>	<i>Effect</i>	<i>Temporary failure</i>		<i>Permanent failure</i>			
		<i>Local effects</i>	<i>Global effects</i>	<i>Local effects</i>	<i>Global effects</i>	<i>Repa-rability</i>	<i>Replacement</i>
<i>Safety-critical control unit</i>		2	2	5	7	n/a	spare truck within 2 hours
<i>Non-safety-critical control unit</i>		1	1	4	5	n/a	spare truck within 2 hours
<i>Built-in TETRA radio set</i>		3	3	4	5	n/a	spare truck within 2 hours
<i>Mobile TETRA radio set</i>		n/a	3	3	5	n/a	spare device within 2 hours
<i>Mobile analogue radio set</i>		n/a	3	4	5	n/a	spare device within 2 hours
<i>Special equipment not needed for mission</i>		1	1	1	2	n/a	n/a
<i>Special equipment needed for mission</i>		1	1	5	6	n/a	spare truck within 2 hours
<i>Permanently mounted emergency light</i>		1	1	2	2	n/a	Later replacement sufficient.
<i>Mobile emergency light</i>		1	1	2	2	n/a	Later replacement sufficient.

Legend: n/a = not applicable.

5.3 Results

Table 4 shows, as an example, the calculated risk priority numbers for the global effects of temporary failures caused by IEMI in the VIP convoy scenario.

Table 4. RPN results example: Temporary failures caused by IEMI in the VIP convoy scenario

Temporary failure	Endangered components									Mean value
IEMI threats	Safety-critical control unit	Non-safety-critical control unit	Built-in TETRA radio set	Mobile TETRA radio set	Mobile analogue radio set	Built-in electronics for special equipment not needed in mission	Built-in electronics for special equipment needed for mission	Permanently mounted emergency light	Mobile emergency light	
Jamming	60	40	120	120	120	10	10	10	10	56
Taser	3	4	6	3	3	1	1	1	1	3
ESD gun	3	2	6	3	3	1	1	1	1	2
HPM	72	64	120	96	96	8	8	16	16	55
UWB	72	64	120	96	96	8	8	16	16	55
LEMP	6	6	12	9	9	1	1	1	1	5
Security attack	9	6	45	45	9	3	3	3	3	14
Mean value	32	27	61	53	48	5	5	7	7	27

The mean values indicate that risks by Jamming, HPM and UWB were considered highest, those by Tasers and ESD guns, in turn, almost negligible. As for vulnerable devices, the radio equipment got the biggest RPN with control units not far behind it. Due to their partly easier accessibility (cf. power mirrors), non-safety-critical control units were ranked almost as risky as safety-critical ones.

As can be seen in Table 5, the mean values for the three scenarios were in a relatively small range, with a minor advantage for the military convoy, which is due to the better hardening of military equipment and the positive image of the military in Austria. Generally, the higher vulnerability of components in respect to temporary failures is widely, if not fully, compensated by the more severe consequences of permanent failures.

As mentioned in section 5.2, the results presented here have been derived from provisional data; for obtaining reliable and up-to-date results, periodic analysis updates by a multi-disciplinary team including the CI operator would be necessary.

Table 5. Mean value of RPN for the three scenarios for temporary / permanent stop

Scenario Failure	Temporary	Permanent	Ø temporary/permanent
VIP convoy	27	29	28
Military convoy	20	19	20
Fire truck convoy	20	31	25
Ø all convoys	22	26	24

6 Conclusions and Further Research

The modified FMEA analysis enabled simplified data capturing and an easy comparison of the average risk for single IEMI weapons as well for single endangered devices. Moreover, the use of general failure modes proved effort-saving.

A weakness of the adapted method is comparability between analysis results of different critical infrastructures. *Qualitative* FMEA sessions by *different expert teams* yield normally different results even when analysing the same critical infrastructure. A more objective and therefore more comparable method seems therefore advisable.

A possible approach to overcome this problem could be to split the estimation of the probability of occurrence extracting criminal energy and political influences on the probability of IEMI attacks as well as availability and cost of IEMI weapons as separate factors. The remaining part of the failure probability is then associated with objective technical conditions comprising local accessibility and electro-technical properties. Based on available EMC research results, the electro-technical vulnerability could be modelled. With this model and a standardized questionnaire, filled in by the CI providers, comparable risk priority numbers should be obtained.

References

1. European Commission, European Programme for Critical Infrastructure Protection EPCIP, http://europa.eu/legislation_summaries/justice_freedom_security/fight_against_terrorism/l33260_en.htm (access on March 8, 2013)
2. Bundeskanzleramt Österreich, Österreichisches Programm zum Schutz Kritischer Infrastruktur APCIP (2008), http://www.kiras.at/uploads/media/MRV_APCIP_Beilage_Masterplan_FINAL.pdf (access on March 8, 2013)
3. Wik, M.W., Radasky, W.A.: Intentional electromagnetic interference (IEMI) – Background and status of the standardization work in the International Electrotechnical Commission (IEC), IEC (2000)
4. EN IEC 61000-4-2, Ed.2, Electromagnetic compatibility (EMC) - Part 4-2: Testing and measurement techniques - Electrostatic discharge immunity test (2008)
5. EN ISO/IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Systems, Ed. 2, Part 1 – Part 7 (2010)
6. ISO/IEC 31010, Risk management - Risk assessment techniques (2009)
7. McMahon, K., et al.: The true cost of road crashes (2012), <http://www.irap.org/about-irap-3/research-and-technical-papers?download=45:the-true-cost-of-road-crashes-valuing-life-and-the-cost-of-a-serious-injury-espaol> (access on October 24, 2012)
8. Guidelines for Hazard Evaluation Procedures; Center for Chemical Process Safety of the American Institute of Chemical Engineers (1992) ISBN 0-8169-0491-X
9. IEC61882, Hazard and operability studies (HAZOP studies) - Application guide, International Electrotechnical Commission (2001)
10. MIL-HDBK-217F. Military Standard, Reliability prediction of electronic equipment. Washington DC: US Department of Defense (1991)
11. TelecordiaTechnologies: Reliability prediction procedure for electronic equipment, Doc. No. SR-332, Issue3 (2011)

A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution

Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson

Department of Computer Science & Engineering, Chalmers University of Technology,
41296, Gothenburg, Sweden

{fatemeh.ayatolahi, behrooz.sangchoolie, roger, johan}@chalmers.se

Abstract. This paper presents the results of an extensive experimental study of bit-flip errors in instruction set architecture registers and main memory locations. Comprising more than two million fault injection experiments conducted with thirteen benchmark programs, the study provides insights on whether it is necessary to consider double bit-flip errors in dependability benchmarking experiments. The results show that the proportion of silent data corruptions in the program output, is almost the same for single and double bit errors. In addition, we present detailed statistics about the error sensitivity of different target registers and memory locations, including bit positions within registers and memory words. These show that the error sensitivity varies significantly between different bit positions and registers. An important observation is that injections in certain bit positions always have the same impact regardless of when the error is injected.

Keywords: out-of-context dependability benchmarking, fault injection, single bit-flips, double bit-flips, error sensitivity.

1 Introduction

Fault injection is an effective and widely used method for test, assessment and dependability benchmarking of fault-tolerant and fail-safe systems. The inclusion of fault injection as a highly recommended assessment method in the recently published ISO 26262 standard [1] for functional safety of road vehicles is an example of the increasing use and importance of fault injection in the embedded systems industry.

One important use of fault injection is out-of-context benchmarking of the error sensitivity¹ of software components. In this type of benchmarking, fault injection experiments are conducted in a test bench that provides an artificial environment to the software component. Out-of-context dependability benchmarking provides a way to build a library of pre-validated software components, which allows the cost of the fault injection experiments to be amortized over several systems and products.

¹ We define error sensitivity as the likelihood that a software component will produce a silent data corruption, as a result of a hardware error.

Two common forms of out-of-context dependability benchmarking of software components are i) robustness testing [2], i.e., testing the ability of a software component to handle input errors, and ii) assessment of the error sensitivity of a software component with respect to hardware faults in the processor and main memory. This paper addresses the latter form of dependability benchmarking.

In general, the ability of a computer system to detect and recover from hardware errors depends on both the hardware architecture and the machine code of the executed software. Hence, when we conduct out-of-context fault injection experiments to assess a software component's ability to handle hardware errors, we must run the experiments on a hardware platform that is similar to the one used in the real product. However, to generalize the benchmarking outcome, different hardware platforms should be considered.

A common objective of such benchmarking is to measure the error sensitivity of a software component, and identify ways to harden the component against such errors by means of software-implemented hardware fault tolerance (SIHFT) techniques.

This type of benchmarking experiments is often conducted by injecting bit-flip errors in main memory words and instruction set architecture (ISA) registers. The injection of such bit-flips is used as an engineering approximation to mimic errors that originate from transistor-level faults in real systems. Examples of such faults include particle-induced single event upsets, and those caused by hardware aging mechanisms such as gate-oxide breakdowns and negative bias temperature instability (NBTI) [3].

In this paper, we present the result of an extensive fault injection study aimed at providing insights into how such bit-flips effect the execution of different programs. To this end, we have conducted a large number of fault injections with thirteen benchmark programs, among which eleven are programs from MiBench suit [4].

The study has two main objectives. First, we investigate differences in the impact of single bit-flips vs. double bit-flips injected in the same target location. (A target location is either an ISA register or main memory word.) This part of the study intends to provide insights to an important open question, namely, whether the single bit-flip model provides optimistic or pessimistic estimates of error sensitivity. Second, we provide statistics about the error sensitivity of target registers and memory words, including individual bit positions.

The first part of our study is partly motivated by the fact that researchers in the field of reliability physics predict that single event upsets (i.e., bit errors caused by strikes of single ionizing particles, such as cosmic neutrons) will be likely to generate multiple-bit upsets (MBUs) in circuits that will become available within a few year from now [5]. While it is still an open question how these MBUs will manifest at the ISA-level in detail, it is clear that we can expect an increasing rate of hardware errors that will manifest as multiple bit errors in main memory words and CPU registers. Although our study only addresses on double bits errors, we believe it provides valuable insights in to the problem of defining multiple-bit fault models for dependability benchmarking experiments.

The remainder of this paper is organized as follows. In Section 2, we describe the target programs and the input sets that are processed by the programs during fault injection. Section 3 describes the fault models and the fault locations of our fault

injection experiments. The experimental setup is explained in Section 4. The results of the experiments are studied in Section 5. In Section 6 we present some discussions. Section 7 describes related work. Finally, we provide conclusions in Section 8.

2 Target Programs

We mainly target programs from the automotive package of the MiBench suite [4] (see Table 1). We select a diverse set of programs with respect to implementation, size, input type/size and functionality, as shown in Table 2. For each program, we choose nine different inputs to represent real applications and to cover all parts of the source code. Since the way a source code programmer (or source code generator) chooses to implement an algorithm changes the executable code — and thus its error sensitivity — we also select different implementations of two algorithms, namely, Bit Count and Quick Sort.

3 Target Locations and Fault Models

The following locations are targeted by our fault injection experiments:

- *Instruction Set Architecture registers (ISA registers)*. General purpose registers (including the stack pointer), the program counter register, and miscellaneous registers (including condition register, link register, and integer exception register).
- *Static Random Access Memory (SRAM) sections*. Stack, data, sdata, bss, and sbss. For simplicity we refer to SRAM locations as memory words in this paper.

Table 1. Target programs

Target Programs	Descriptions
Cyclic Redundancy Check(CRC)	This algorithm is a software implementation of the well known 32-bit cyclic redundancy check used in the Ethernet protocol. This program is from the telecomm package of MiBench suite.
Secure Hash Algorithm (SHA-1)	This algorithm generates a 160-bit digest from inputs. It is widely used in security protocols such as SSL and SSH. The MiBench implementation of SHA-1 uses dynamic memory allocation, which is rarely used in automotive embedded systems. Since our research focuses on such systems, we use an implementation that avoids dynamic memory allocation.
Quick Sort	We use three different quick sort programs, each corresponding to a different implementation of the well known quick sort algorithm. The first corresponds to the original MiBench implementation in which the input set is an array of string elements; it uses the built-in C language function qsort. The second changes the input set to be an array of integers. The third is a recursive implementation of the quick sort algorithm.
Binary to Integer	This program converts a binary number encoded as a string of “0” and “1” characters into its equivalent integer value.
Bit Count	This algorithm counts the number of ones in the binary representation of its input. We use five different implementations of this algorithm.
Square Root Calculator	This program calculates the square root of the input.
Cubic Equation Calculator	This program calculates roots of a cubic equation using floating point arithmetic implemented in software.

Table 2. Size and input range of each target program

Program	Size (bytes)	Input Range
CRC	644	Strings of 0 to 99 characters.
SHA1	3325	
QSort_Mib_Stct	3932	Arrays of 0, 10, 20, 30, and 40 string elements that are half sorted.
QSort_Mib_Int	3228	Permutations of 6 integer elements from totally sorted to totally unsorted.
QSort_Int	964	
BinToInt	959	Binary strings of 0 to 31 characters.
BitCnt1	540	Variables of type long with 0 to 31 1's in their binary format.
BitCnt2	680	
BitCnt3_BW	592	
BitCnt3_AR	684	
BitCnt4	576	
Isqrt	652	Unsigned long variables selected to ensure variations in the number of executed machine instructions.
Cubic	27472	Coefficients of type double chosen to ensure variations in the number of executed machine instructions.

We use *single bit-flips* in our first set of fault injections, where we randomly flip one bit from a target ISA register or memory word. In our second set of fault injections, we use *double bit-flips* as one variation of multiple-bit upset, where we randomly flip two bits within one ISA register or memory word.

4 Experimental Setup

Programs under test are executed on a PowerPC-based microcontroller from Freescale (MPC565). Our fault injection tool, Goofi-2 [6], uses a debugger with a NEXUS [7] interface to inject faults into the ISA registers and memory.

We define a fault injection *experiment* to be the injection of one fault (either a single bit-flip or double bit-flip, according to the fault model) and the monitoring of its impact on the program. A fault injection *campaign* is a set of fault injection experiments using the same fault model on a given workload.

Goofi-2 defines faults as time-location pairs according to a fault-free execution of a workload. Here locations are randomly selected bits to be flipped from the ISA registers or the memory words, and time is a point in the execution trace.

To avoid experiments that would not have an observable impact on the program, we employ Barbosa et al.'s analysis [8] to exclude unreachable locations from the fault space; in other words, faults are only injected in a target location immediately before the location is read by an instruction.

5 Experimental Results

For each of the 13 target programs, we conducted nine campaigns with the single bit-flip and double bit-flip models. Each campaign consists of 12,000 experiments running a program on a fixed set of inputs. In total, the study comprises 2,808,000 experiments. The corresponding 95% confidence intervals are between $\pm 0.06\%$ and $\pm 0.3\%$. The outcome of each experiment is classified into one of the following categories:

- *No Impact*. The program terminates normally and the error does not affect the output of the program.
- *Hardware Exception*. The processor detects an error by raising a hardware exception.
- *Timeout*. The program fails to terminate within a predefined time (which is set to be approximately 10 times larger than the execution time of the workload).
- *Silent Data Corruption (SDC)*. The program terminates normally, but the output is erroneous and there is no indication of failure.

5.1 Single Bit Flips vs. Double Bit Flips

Here we compare differences in the impact of single bit flips and double bit flips, including the error sensitivity for different target registers and memory locations.

5.1.1 Average Results of Single and Double Bit Flips

Fig. 1 shows the average results over all campaigns for single bit flips, Fig. 1 (a), and double bit flips, Fig. 1 (b). The most noticeable differences between the two fault models are in Hardware Exception and No Impact categories. While Hardware exceptions are eight percentage points higher for double bit flips, the percentage of experiments in the No Impact category is six percentage points lower for the double bit flips. The difference for the SDC category is only two percentage points. The percentage of SDC is lower for the double bit-flip model for approximately 75% of the campaigns. For all of these campaigns, the difference between the two fault models is around three percentage points. However, for the remaining 25% of the campaigns, we observe a difference of around only half a percentage point between the two fault models, on average. There is no significant difference between the two fault models for the Timeout category. In fact, for approximately 90% of the campaigns the difference is less than one percentage point.

Table 3 shows variations in the fault injection outcomes for different programs. For example, for the single bit-flips, QSort_Mib_Stct and SHA1 have the lowest (16%) and the highest (42.4%) SDCs, respectively. Whereas QSort_Int and Isqrt have the lowest (15.8%) and the highest (39.7%) SDCs for the double bit-flips.

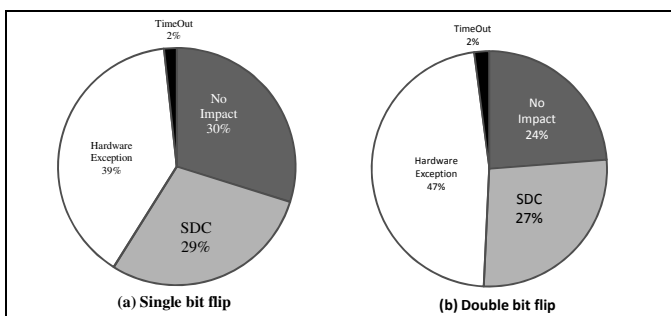


Fig. 1. Average outcome distributions over all programs

Table 3. Average outcome distributions for different programs

	No Impact (%)		Silent Data Corruption (%)		Hardware Exception (%)		TimeOut (%)	
	Single	Double	Single	Double	Single	Double	Single	Double
CRC	23.4	15.7	31.2	30.1	44.4	52.9	1.1	0.6
SHA1	15.8	10.3	42.4	38.1	40.3	50.0	1.5	1.4
QSort_Int	30.4	24.0	19.2	15.8	46.3	55.9	4.1	4.3
QSort_Mib_Int	34.3	29.4	18.0	16.5	46.6	52.7	1.2	1.3
QSort_Mib_Stct	38.3	33.9	16.0	16.1	43.3	48.3	2.3	1.7
BinToInt	35.9	31.5	20.3	19.0	41.0	47.0	2.8	2.5
BitCnt1	23.0	22.7	39.1	28.0	32.1	39.1	5.9	10
BitCnt2	31.8	21.7	33.1	34.5	35.0	43.7	0.2	0.1
BitCnt3_AR	27.6	18.8	34.5	34.1	37.7	47.0	0.2	0.1
BitCnt3_BW	35.2	27.5	26.5	25.4	38.0	47.0	0.2	0.1
BitCnt4	21.5	15.9	31.0	26.8	46.0	54.7	1.4	2.5
Isqrt	27.5	21.8	41.3	39.7	30.7	37.6	0.6	0.8
Cubic	43.0	35.3	25.8	25.9	30.0	37.6	1.1	1.1

Comparing the results of different workloads, we observe that there is a higher variation in the double bit-flips for the Hardware Exception, while the single bit-flips show higher variation in the No Impact and SDC categories. In order to find the reasons behind variations of results in the two fault models, in the next section, we analyze how different target locations contribute to each outcome category.

5.1.2 Comparing Single and Double Bit-Flips According to Fault Locations

Fig. 2 (a) shows that the program counter register is the main contributor to the Hardware Exception. In average, more than 73% and 92% of the injections in the program counter register are detected by the hardware exceptions for single and double bit-flips, respectively. The majority of the detections are due to attempts to execute instructions that are not implemented or accesses to illegal addresses.

With respect to the injections in general purpose registers and memory (Fig. 2 (b) and (c)), in general, the results are biased towards generating SDC and No Impacts. This is due to the nature of general purpose registers and memory words where they can be used to store data or addresses. Generally, errors in address values are more probable to be detected by hardware exceptions, while errors in data values seem to be more difficult to be detected. Therefore, injecting faults in locations holding data values are more probable to be classified as SDC or No Impact. An interesting observation here is the variation in the outcomes of different bit count implementations. Even though, the variation is not significant for program counter register, we see significant variations in the outcomes for general purpose registers and memory.

5.2 Error Sensitivity for Bit Positions within Registers and Memory Words

In this section, we present results regarding the error sensitivity of bit positions in the program counter, stack pointer, general purpose registers and main memory locations. All results are presented as aggregated data over all target programs. For general purpose registers and main memory words, the data is also aggregated over all registers and memory words used during program execution.

We enumerate the bits 1 to 32 and present outcome distributions for each bit position. All data presented in this section are from injections of single bit errors. One reason for conducting this study is to understand some of the reasons behind the differences between the outcomes of the single and double bit errors that we observed in the previous section. The number of injections in each bit varies from around 2,000 up to almost 18,000 depending on the target register or memory, see Table 4. Hence, the statistical confidence of the results is fairly high.

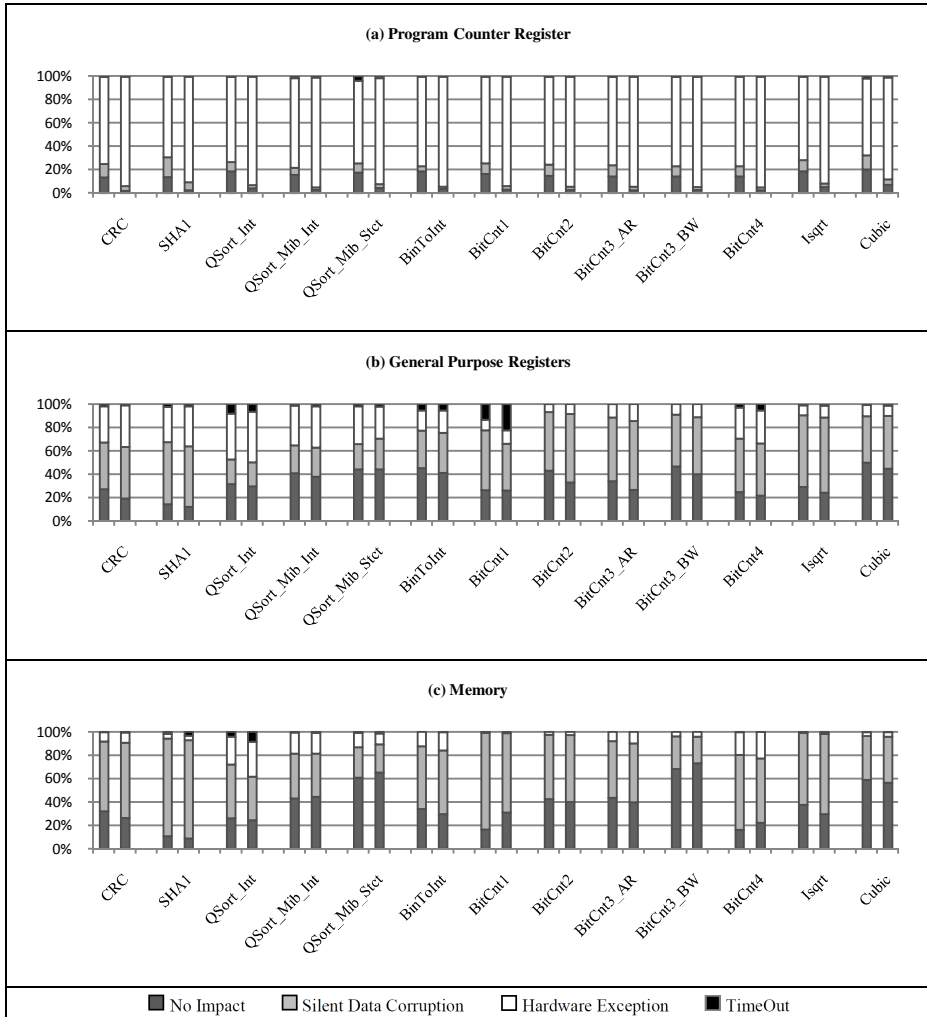


Fig. 2. Outcome distributions for different programs. For each program: Left bar: *Single bit-flips*, Right bar: *Double bit-flips*.

Table 4. The average number of fault injections (per bit) for target registers and memory

Target Location	Average number of fault injections (per bit)
Program Counter Register	17037
Stack Pointer Register	1942
General Purpose Registers	17842
Memory	4631

Program counter register (Fig. 3 (a)). We see that injections in bit 1 and 2 in almost all cases have no impact. The reason for this is that the PowerPC architecture does not use these bits when the processor fetches instructions from main memory. With respect to the errors injected in bits 3 to 16, we see that the percentage of errors detected by hardware exceptions increases with increasing bit numbers, while the percentage of No Impact experiments decreases. Every error injected in bits 17 to 32 is detected by hardware exceptions.

In general, the first (least significant) bit that is fully covered by hardware exceptions is highly dependent on the program size. For example, for *BitCnt1*, which is the smallest program, all errors injected in bit 11 and higher are detected by hardware exceptions. For *Cubic*, which is the largest program, all errors injected in bit 17 and higher are detected by hardware exceptions.

Stack pointer (Fig. 3 (b)). For the stack pointer, results vary between different bits, except for bits 17 to 22. For these six bits, all injected faults are detected by hardware exceptions. This can be explained by studying the internal memory block of our target setup. The internal memory is 4Mbytes that resides in 0x0000 0000 to 0x003F FFFF address block and the SRAM is located from the address 0x3F7000 to 0x3FFFFFFF. The stack pointer always contains an address to the SRAM area. Therefore, all the addresses referring to the SRAM contain 1 in bits 17 to 22. Flipping any of these bits from 1 to 0 will result in an address smaller than the SRAM base address which triggers hardware exceptions. We also expected accesses above the SRAM upper bound to be detected by hardware exceptions. However, as we can see the errors in bits 23 to 32 are not always detected by hardware exceptions. The reason for this behavior is likely to be related to implementation of the address decoding logic on the processor board that we use for our experiments.

General purpose registers and memory (Fig. 3 (c) and (d)). The outcome distributions for general purpose registers and memory follow a similar trend. These locations typically store either data or addresses. If they hold an address, the impact of an error is likely to be similar to the impact of errors in the stack pointer, where errors in bit 17 to 22 always raise a hardware exception. (Note that the results for the stack pointer register (R1) are not included in this data.)

We see a similar result for the general purpose registers, where the proportion of errors detected by hardware exceptions is significantly higher for bits 17 to 22 compared to other bit positions. This trend is also seen for the memory words, although it is less pronounced compared to the general purpose registers. This result suggests that the general purpose registers more often holds addresses than the main memory words do. This observation is also supported by the fact that silent data corruption is the dominating outcome for all bit positions in the memory words.

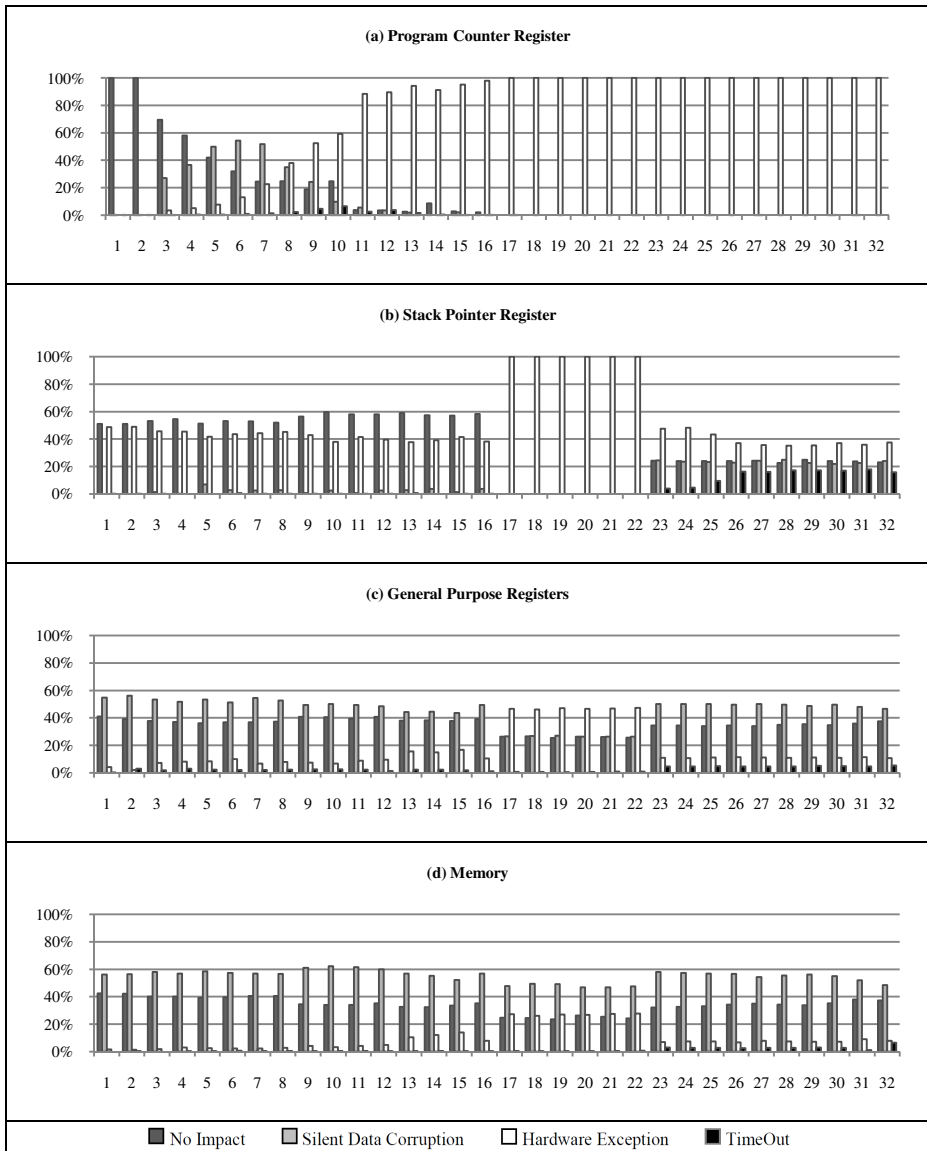


Fig. 3. Outcome distributions over the 32-bits registers and memory

6 Discussion

In the first part of our results, we saw that the main difference in the impact of single bit errors and double bit errors was in the proportions of the outcomes No Impact and Hardware Exception. The double bits errors had a much higher percentage of errors detected by hardware exceptions, and a lower percentage of errors having no impact or causing silent data corruption, compared to the single bit errors.

This result can partially be explained by the observations we made in the second part of the study where we looked at the error sensitivity of individual bits in different target locations. When we look at results presented in Fig. 3, it is clear that we are likely to increase the chance of detecting an error by means of a hardware exception if we increase the number of bit-flips injected in the target locations. This is especially obvious for the stack pointer and the program counter register, where bit-flips in certain bit positions are always detected by a hardware exception.

Furthermore, we see a potential to improve the pre-injection analysis by avoiding the injections in high significant bits of program counter depending on the size of the program. Also, the two least significant bits can be removed due to the high percentage of experiments resulted in No Impact. In addition, we can exclude bits 17 to 22 of the stack pointer due to full detection by hardware exceptions.

In favor of the effectiveness of our fault injection assessment, the pre-injection analysis removes several miscellaneous registers bits from the experiments. Thus we ignore these registers from our analysis. However, it is worth noting that the link register, which contains return address of a function, almost shows a similar trend as the program counter register. Also with respect to the condition register, in some programs, the injections are distributed over all bits, whereas in others, the focus has been given to some specific bits.

7 Related Work

Various fault injection tools have been developed in the past decades to assess dependability properties of computer systems. Popular fault injection techniques include pin level injection [9], software implemented [10], fault injection via debug interfaces such as Nexus [11] [6], hardware implemented [12], and simulation-based [13].

The results of fault injection experiments depend on several parameters such as the inputs processed by a program [14] [15], level of compiler optimization [16], fault model [13], implementation of a program, etc. In this paper we study the impact of single bit-flip and double bit-flip fault models.

The impact of device-level faults which manifest as single bit-flips in the ISA registers and main memory has been studied in literature [14] [17]. Some recent studies have targeted SRAMs and DRAMs to multiple-bit upsets (MBUs) [18] in order to investigate geometric effect of MBU faults. In addition, the authors of [13] investigated the impact of single/multiple bit-flips in the LEON2 processor using fault injection in a VHDL simulation model. We study another level of abstraction where we mimic bit-flips in ISA registers and memory of a real hardware platform. Although there are some fundamental differences between our work and theirs, they also observed that the percentage of No Impact experiments is higher for the single bit-flip model, while the percentage of Hardware Exception experiments is lower for the single bit-flip model. However, they showed that the percentage of experiments classified as SDC was higher for the double bit-flip model, which is not the case in at least 75% of our campaigns. In fact, we observed that in some campaigns, the percentage of experiments classified as SDC is 15 percentage points lower for the double bit-flip model compared to single bit-flip model.

8 Conclusions and Future Work

We have presented results from an extensive fault injection study that investigates the impact of bit-flip errors in ISA-registers and main memory for thirteen programs. The purpose of the study is to provide insights into differences in impact between single-bit and double-bit errors. The intended audience for this research is individuals and organizations who are interested in experimental benchmarking of the error sensitivity of software components.

Such benchmarking experiments aim to measure the likelihood that the executable code of a software component exhibit silent data corruptions (SDCs) for hardware errors that propagate to instruction set architecture (ISA) registers and main memory locations. The purpose of such measurements is to identify weaknesses in the executable code, and thereby finding ways of hardening the code against hardware errors by means of software-implemented hardware fault tolerance techniques.

This type of benchmarking is traditionally conducted by injection of single-bit errors in ISA registers and main memory words. The main objective of the study was to investigate whether it would be meaningful to include experiments with double bit errors when measuring the error sensitivity of software components. The inclusion of double bit-flip injections would be motivated if we can see that such errors would reveal weaknesses that are not exposed by single bit-flip injections.

Based on the discussion in Section 6, we believe the use of double and multiple bit injections mainly would lead to fewer observations of silent data corruptions. This suggests that it is unlikely that experiments with double-bit errors would expose weaknesses that are not revealed by single bit-flips injection. To further assess whether single bit flips can be trusted to generate the most pessimistic results (the highest number of SDCs), our future work will include experiments where bit-flips will be injected in different target locations at the same time.

In the second part of the study, we investigated, for single bit-flips, how different bit positions contributed to each of the four outcome categories (No Impact, Hardware Exception, Timeout, and Silent Data Corruption). This analysis gave us useful insights that can help us reduce the fault space of future experiments through a more elaborate strategy for selecting target locations. One such observation was that bit-flips in certain bit positions in registers holding addresses always were detected by a hardware exception. This suggests that the memory map of the target system should be considered in the pre-injection analysis.

Our future research will also encompass comparative studies with different compiler optimizations, hardware platforms, and different programming languages. Another important part of this work is to extend our study with experiments on target programs that are equipped with software-implemented hardware fault tolerance (SIHFT) techniques.

References

1. ISO Standard, http://www.iso.org/iso/catalogue_detail?csnumber=43464 (accessed 2012)
2. Kropp, N.P., Koopman, P.J., Siewiorek, D.P.: Automated robustness testing of off-the-shelf software components. In: 28th Annual Int. Symp. on Fault-Tolerant Computing (June 1998)

3. Schroder, D.K.: Negative bias temperature instability: What do we understand? *Microelectronics Reliability* 47(6), 841–852 (2007)
4. MiBench Version 1.0, University of Michigan, <http://www.eecs.umich.edu/mibench/>
5. Suh, J.: Models for Soft Errors in Low-level Caches. University of Southern California (May 2012)
6. Skarin, D., Barbosa, R., Karlsson, J.: GOOFI-2: A tool for experimental dependability assessment. In: *IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN (2010)*
7. Nexus 5001TM Forum, IEEE-ISTO (1999), <http://www.nexus5001.org/>
8. Barbosa, R., Vinter, J., Folkesson, P., Karlsson, J.M.: Assembly-level pre-injection analysis for improving fault injection efficiency. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) *EDCC 2005. LNCS*, vol. 3463, pp. 246–262. Springer, Heidelberg (2005)
9. Madeira, H., Rela, M.Z., Moreira, F., Silva, G.J.: RIFLE: A general purpose pin-level fault injector. In: Echtle, K., Powell, D.R., Hammer, D. (eds.) *EDCC 1994. LNCS*, vol. 852, pp. 197–216. Springer, Heidelberg (1994)
10. Kanawati, G., Kanawati, N., Abraham, J.: FERRARI: a tool for the validation of system dependability properties. In: *22nd Int. Sym. on Fault-Tolerant Computing, FTCS-22 (July 1992)*
11. Yuste, P., Ruiz, J.C., Lemus, L., Gil, P.: Non-intrusive Software-Implemented Fault Injection in Embedded Systems. In: de Lemos, R., Weber, T.S., Camargo Jr., J.B. (eds.) *LADC 2003. LNCS*, vol. 2847, pp. 23–38. Springer, Heidelberg (2003)
12. Fidalgo, A.V., Alves, G.R., Ferreira, J.M.: Real Time Fault Injection Using Enhanced OCD – A Performance Analysis. In: *21st IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT 2006) (October 2006)*
13. Touloupis, E., Flint, J.A., Chouliaras, V.A., Ward, D.D.: Study of the Effects of SEU-Induced Faults on a Pipeline Protected Microprocessor. *IEEE Transactions on Computers* 56(12), 1585–1596 (2007)
14. Di Leo, D., Ayatolahi, F., Sangchoolie, B., Karlsson, J., Johansson, R.: On the Impact of Hardware Faults - An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions. In: Ortmeier, F., Lipaczewski, M. (eds.) *SAFECOMP 2012. LNCS*, vol. 7612, pp. 198–209. Springer, Heidelberg (2012)
15. Segall, Z., et al.: FIAT-fault injection based automated testing environment. In: *18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pp. 102–107 (1988)
16. Demertzi, M., Annavaram, M., Hall, M.: Analyzing the Effects of Compiler Optimizations on Application Reliability. In: *IEEE Int. Symp. on Workload Characterization (IISWC)*, Austin, TX (2011)
17. Skarin, D., Karlsson, J.: Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System. In: *7th European Dependable Computing Conference (EDCC 2008) (May 2008)*
18. Satoh, S., Tosaka, Y., Wender, S.A.: Geometric Effect of Multiple-Bit Soft Errors Induced by Cosmic Ray Neutrons on DRAM's. *Electron Device Letters* 21(6), 310–312 (2000)

OpenMADS: An Open Source Tool for Modeling and Analysis of Distributed Systems

Ermeson C. Andrade, Marcelo Alves, Rubens Matos,
Bruno Silva, and Paulo Maciel

Federal University of Pernambuco
Recife, PE, Brazil

{ecda, mma2, rsmj, bs, prmm}@cin.ufpe.br
<http://www.modcs.org>

Abstract. In this paper, we present OpenMADS, an open source tool for modeling and analysis of distributed systems. OpenMADS generates comprehensive availability models by using the input of SysML specifications and MARTE annotations, which are automatically translated into deterministic and stochastic Petri nets. The integrated use of analytic models (e.g., Petri Nets or Markov chains) with semi-formal modeling languages, like SysML or UML, can provide important insights to the designers regarding different distributed infrastructures, and consequently, allows them to choose the infrastructure that fits the company budget or satisfies a given service level agreement. To show the applicability of OpenMADS, we demonstrate the process of availability modeling and evaluation based on the example of a Web server system.

Keywords: Distributed Systems, MARTE, SysML and DSPN.

1 Introduction

Over recent years, distributed systems such as cloud computing and grid computing have grown significantly in functionality, scale and complexity. Although this rapid growth has allowed these systems to provide a wide range of services as well as has increased the number of users, it has also increased the occurrence of system failures due to high complexity and the existence of strongly coupled components. Therefore, modeling and evaluation of such computing systems are important steps in the design process of distributed systems.

Analytic models such as Markov chains and stochastic Petri nets are useful to study a wide range of systems, but they are not easy to use by designers who do not have expertise in stochastic modeling. An important and challenging issue is to enable designers to develop analytic models representing distributed system configurations and behaviors. In order to address this issue, we advocate the use of Systems Modeling Language (SysML) [13] and MARTE (UML profile for Modeling and Analysis of Real-Time and Embedded systems) [14] to generate Deterministic and Stochastic Petri Nets (DSPNs) [5]. SysML is used for various engineering design purposes and supports more friendly and intuitive

notation methods. However, SysML itself does not provide support to quantitative notations. Quantitative notations are especially important when modeling performance/availability concepts of distributed systems. Thus, we adopted the combination of SysML and MARTE annotations for a complete design of distributed systems. Still the combination of SysML diagram and DSPNs needs an appropriate tool to be practicable.

There are several commercial and open source tools which support the use of SysML diagrams, such as Papyrus [16] for SysML (open source eclipse modeling tool), ARTiSAN Software Tools [15] and ParaMagic (InterCAX) [11]. Besides that, many other tools have been developed to support the evaluation of analytic models. Examples include: SPNP [10], TimeNET [3] and ASTRO [4]. Some tools also have been developed to support the integrated use of semi-formal models and analytic models, e.g., ADAPT [1], LSC2CPN [12] and ArgoSPE [6]. However, to the best of our knowledge, there are no tools which allow the integrated use of SysML diagrams, MARTE annotations and DSPNs for modeling and analysis of distributed systems.

In this paper, we present OpenMADS, an open source tool for modeling and analysis of distributed systems, considering dependability aspects. It extends and implements the features given in our previous work [9,7]. So, the distributed system is designed using SysML diagrams, annotated according to the MARTE profile, which are automatically translated into DSPNs. After that, the model can be evaluated to compute a set of availability measures such as system steady-state availability and downtime. The remainder of the paper is organized as follows: Section 2 presents the software architecture of OpenMADS. Section 3 details an example and its results. Section 4 concludes the paper and discusses further work.

2 Architecture of OpenMADS

OpenMADS has been developed to support both the translate process from SysML diagrams to DSPNs and their evaluation. This tool allows designers, who do not have expertise in formal model, to design and analyze distributed systems on a cloud computing platform. The designers can use OpenMADS to: (i) design the system infrastructure using SysML diagrams and MARTE annotations, (ii) generate the availability model through the translation process, and (iii) study different distributed system infrastructures. It allows designers, for example, to choose the service infrastructure that fits their budget or satisfies a given Service Level Agreement (SLA). OpenMADS can be accessed in [8].

Figures 1 and 2 show screenshots of OpenMADS GUI. The graphical user interface for OpenMADS has been completely written in Java, and therefore, can run in both Unix- and Windows-based environments. It is composed of three main parts: a menu bar (top), drawing area (bottom), and a SysML and MARTE annotation bar or a stochastic Petri Net bar (middle). The upper row of the window contains some menus with commands for file handling, editing, and other model specific commands. A toolbar at the middle contains model

elements that are available for the current SysML diagram or stochastic Petri net model. The drawing area at the bottom displays the current model. The main functionalities implemented in the tool are detailed as follows.

- **SysML and MARTE editor:** Designers can build models for distributed system configurations and maintenance operations using the draw area of OpenMADS. It supports three SysML diagrams: internal block (SysML-IBD), state machine (SysML-STM) and activity (SysML-AD). SysML-IBD is used to describe the static system configuration of distributed systems, such as logical function, process structure, hardware with and without their redundancies. SysML-STM describes the state transitions of a specific system element, e.g., a server failure-recovery behavior. SysML-AD describes the process flow of administrative operations which may affect the system state (e.g., backup, server restart, etc). OpenMADS also provides MARTE annotation which can be assigned to states and transitions. One should note that only a subset of MARTE annotation is provided by our tool. The stereotype *PASep* and tagged value *HostDemand* are used. The stereotype describes an action, while tagged values consist of a property name and an assigned value.
- **DSPN editor and Simulator:** Regarding DSPN models, OpenMADS allows dependability evaluation utilizing simulation techniques, such as transient and stationary. Time-dependent metrics are obtained through transient simulations, while steady-state metrics are result of stationary simulations. Figure 2 depicts the DSPN editor, in which the models can be obtained from a high level model translation or created by the user from scratch.
- **ASTRO-Mercury and TimeNET integration:** In order to allow various analysis and simulations of the DSPNs, considering distinct levels of accuracy, we have integrated OpenMADS to ASTRO-Mercury [4] and TimeNET [3] tools. That is, OpenMADS allows designers to generate input files for these tools. ASTRO-Mercury and TimeNET are analytic modeling tools that provide user-friendly graphical interfaces for hybrid models which include Petri Nets, Reliability Block Diagrams (RBDs) and Data Center High-Level models.

Figure 3 presents the software architecture of OpenMADS. From the SysML diagrams, annotated according to the MARTE, OpenMADS creates a parameterized XML file that will be an input for the *Translator Module*. The *Translator Module* implements the translation process from the parameterized XML model into the DSPN formalism. OpenMADS implements the translation theory proposed in [9,7]. In the translation process, each element in SysML diagrams is translated into a part of the availability model, and these parts are assembled and synchronized together to construct the whole availability model in the form of DSPNs. The resulting DSPN model XML file is used to perform availability analysis through transient and stationary simulation techniques. The results of such availability evaluation are displayed in the GUI. For DSPN models, OpenMADS implements an *Integrator Module*, which generates input files for

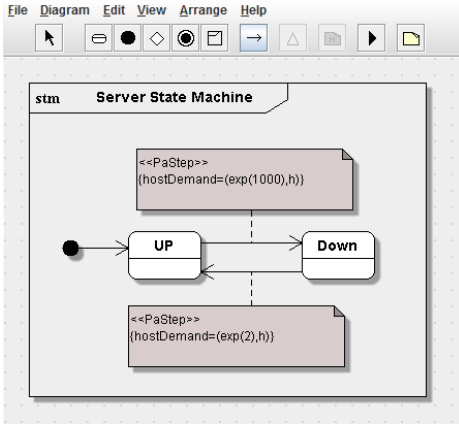


Fig. 1. SysML and MARTE editor

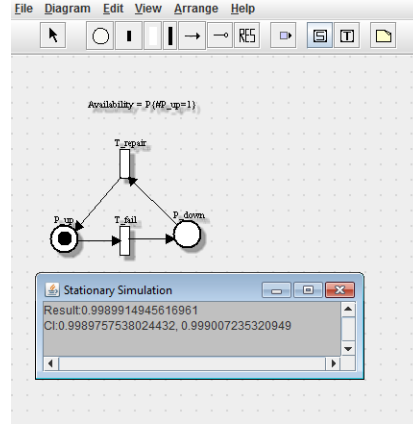


Fig. 2. DSPN editor

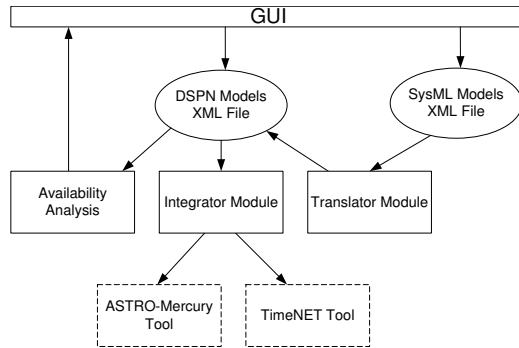


Fig. 3. Software architecture

ASTRO-Mercury and TimeNET tools. It is worth to highlight that OpenMADS is not integrated with other SysML modeling tools, but it can be done by implementing a module that adapts the parameterized XML file as input file for the target tool.

3 An Illustrative Example

In order to show the applicability of OpenMADS, we present an illustrative example of a Web server system composed of one load balance server, six Web application servers, and one database server deployed in a Data Center (see Figure 4). This example is a typical virtualized distributed system in which each server runs on its own Virtual Machine (VM) and shares the resource of the Data Center. A load balance server distributes the user requests evenly among the Web application servers, avoiding that a single VM becomes overloaded

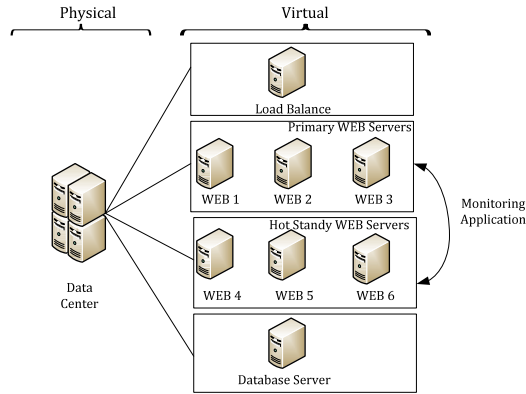


Fig. 4. Web server system

while the others are idle. There are six virtual machines running the same Web application. Three of those VMs are in Hot Standby state. If a primary Web application server stops, a Hot Standby Web server replaces the one which is no longer available. Once the primary Web server is repaired, then a working server returns to the Hot Standby state. We assumed that the minimum number of running Web servers is 3 and it is maintained by an automatic monitoring application. If a Web server become unavailable, the monitoring application automatically increases the number of Web servers up to 3 in order to meet the performance requirements. Besides, the Web application servers access a Database server running in the same Data Center to store information.

Due to the lack of space, the SysML diagrams and the corresponding DSPNs are briefly detailed. Figure 5 (e) presents the SysML-IBDs for the Web server system deployed in a Data Center. The number "6" in the *Web Server* block means the level of redundancy of the system component. Multiple standard compartments can be used to describe the block characteristics. The *Load Balancer* has the Mean time to Failure (MTTF) of 8760 hours, and the Mean Time to Repair (MTTR) of 120 hours based on the block properties. The *allocatedFrom* and *hosted* allocations are used to represent relationships among the SysML diagrams. An *allocatedFrom* allocation between a block and a SysML-STM is used to detail the block in terms of states, while an *allocatedFrom* allocation between a block and a SysML-AD is used to represent administrative operations (i.e.: rejuvenation, backup or restart) which may affect the states of a system element. A *hosted* allocation represents a hosted dependency between system elements. If the hosting element (Data Center) goes down, the hosted elements (Load balance, Web Servers and Database Server) becomes unavailable at the same time.

The blocks *Data Center*, *Web Server* and *Database Server* are respectively detailed in terms of state transitions in Figures 5 (b), (c) and (d). Note that *stm Database State Machine* and *stm Data Center State Machine* have the same

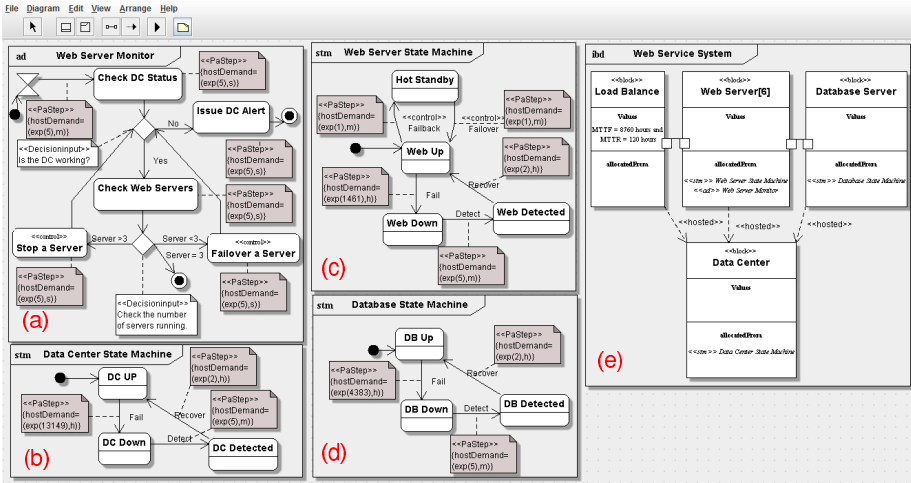


Fig. 5. SysML diagrams for the Web server system

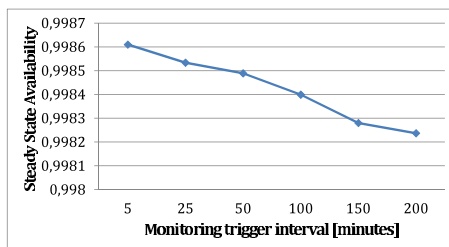
failure/repair behavior. However, they have different time restrictions assigned to the transitions (MARTE notations). The failure/repair behavior of the *Web Server* is also similar to the ones in Figures 5 (b) and (d). The difference for the SysML-STM is that there is a *Hot Standby* state and two SysML transitions connecting the *Web UP* state to the *Hot Standby* state (failover and fail-back operations). The SysML transitions *Failover* and *Failback* are stereotyped with $\ll\text{control}\gg$. That means an administrative operation (i.e., *ad Web Server Monitor*) may affect the system states.

The SysML-AD used to represent the behavior of the *Web Server Monitor* is depicted in Figure 5 (a). This administrative operation is deployed in the *Web Servers*. The *Web Server Monitor* checks the data center every 5 minutes. If the data center is not working properly, an alert message is issued to system administrators who are responsible for manual maintenance operations. Otherwise, the number of available Web servers is checked and it is adjusted, if necessary. We introduced the stereotype $\ll\text{control}\gg$ to the actions *Failover a Server* and *Stop a Server*. That is, the *Web Server* states (see Figure 5(c)) are affected by the execution of the action nodes *Stop a Server* and *Failover a Server* of the respective SysML-AD.

The following procedure is adopted to obtain the DSPN models from the SysML diagrams depicted in Figure 5. First, OpenMADS translates the SysML diagrams into parts of the availability model called *model components*. Next, OpenMADS assembles the model components generated from SysML-IBD and SysML-STM using the allocation notation or block properties. The resulting DSPN is called *System Net*. Finally, OpenMADS synchronizes the *System Net* and *Activity Net* (which is a model component generated from a SysML-AD) and supports the generation of guard functions. Note that the DSPNs models

Table 1. Steady-state availability and downtime

	Availability	Downtime
Load Balance	0.9998313	1.477812
Web Server	0.9989158	9.497242
Database Server	0.9993347	5.828028
Data Center	0.9996610	2.969640
System (A_{sys})	0.9986098	12.178152

**Fig. 6.** Sensitivity analysis

are obtained by simple clicking in the translation button. These models were omitted due to lack of space. Details regarding the applicability and correctness of the translate process can be found in [9,7,17]. However, it is important to highlight that OpenMADS extends our previous work by taking into account both dependability constraints described through MARTE annotations and different types of allocations.

OpenMADS was used to compute system availability of the DSPNs generated by the translation process. We use arbitrary (but reasonable) input parameters, since the purpose of this example is to show the applicability of OpenMADS. Note that all these parameters are defined through MARTE annotations and block properties and they are assumed to be exponentially distributed except by the monitoring trigger interval to check the Data Center status, which is deterministic. The availability and downtime (hours per year) of the Web Server system and each of its components are shown in Table 1. One should note that A_{sys} is not equal to the product of the availability for each component because they are not independent. The results show that the Web Servers have the lowest availability among all components in the system, being therefore one of the dependability bottlenecks in the modeled environment.

We conducted sensitivity analysis by varying the time interval of the *Web Server Monitor*. As it can be seen from Figure 6, if the monitoring trigger interval is close to zero, the Web servers are checked constantly avoiding downtime, and consequently, yields higher system availability. On the other hand, as the monitoring trigger interval increases, the system availability drops, since outages of the Web servers may take longer to be detected. The annual downtime increases from 12 hours to 15.5 hours, if we change the monitoring interval from 5 minutes to 200 minutes. This is an example of many important conclusions which can be obtained by using OpenMADS to model distributed systems.

4 Conclusions

We have presented OpenMADS, an open source tool for modeling and analysis of distributed systems. It is not easy for designers to make a comprehensive availability model from scratch. The use of tools which support the modeling and

evaluation of distributed systems is essential. OpenMADS generates an availability model by using the input of SysML descriptions and MARTE annotations. The proposed tool translates the elements of SysML diagrams, annotated according to the MARTE profile, into DSPNs. After that, the model is evaluated to compute a set of availability measures in the earliest stages of the design process of distributed systems. We have demonstrated the process of availability modeling and evaluation based on the example of a Web server system. In future work, we plan to model and evaluate other types of distributed systems.

References

1. Rugina, A.-E., Kanoun, K., Kaâniche, M.: The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. In: EDCC (2008)
2. Team, A.: The ArgoUML Tool (2013), <http://argouml.tigris.org/>
3. Zimmermann, A., Knoke, M., Huck, A., Hommel, G.: Towards Version 4.0 of TimeNET. In: MMB,, pp. 1–4 (2006)
4. Silva, B., Callou, G., Tavares, E., Maciel, P., et al.: Astro: An Integrated Environment for Dependability and Sustainability Evaluation. In: Sustainable Computing: Informatics and Systems (2012)
5. Marsan, M., Chiola, G.: On Petri Nets with Deterministic and Exponentially Distributed Firing Times. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 132–145. Springer, Heidelberg (1987)
6. Gómez-Martínez, E., Merseguer, J.: ArgoSPE: Model-Based Software Performance Engineering. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 401–410. Springer, Heidelberg (2006)
7. Andrade, E., Machida, F., et al.: Modeling and Analyzing Server System with Rejuvenation Through SysML and Stochastic Reward Nets. In: ARES (2011)
8. Andrade, E., Alves, M., Maciel, P.: OpenMADS: An Open Source Tool for Modeling and Analysis of Distributed Systems (2013), <http://code.google.com/p/openmads/>
9. Machida, F., Andrade, E., et al.: Candy: Component-Based Availability Modeling Framework for Cloud Service Management Using SysML. In: SRDS (2011)
10. Ciardo, G., Muppala, J., Trivedi, K.: SPNP: Stochastic Petri Net Package. In: Proceedings of the Third International Workshop, PNPM, pp. 142–151 (1989)
11. InterCAX, Paramagic - SysML Parametric Solver for MagicDraw (2013), <http://www.intercax.com/sysml>
12. Amorim, L., Maciel, P., Nogueira, M., Barreto, R., Tavares, E.: Mapping Live Sequence Chart to Coloured Petri Nets for Analysis and Verification of Embedded Systems. ACM SIGSOFT Software Engineering Notes 31 (2006)
13. Hause, M., et al.: The SysML Modelling Language. In: Fifteenth European Systems Engineering Conference, vol. 9 (2006)
14. Faugere, M., et al.: Marte: Also an UML Profile for Modeling AADL Applications. In: ICECCS (2007)
15. Hause, M., Gloucestershire, C., Hn, G.: Artisan Software Tools (2013), <http://www.atego.com/products/artisan-studio/>
16. Gérard, S.: Papyrus UML (2013), <http://www.papyrusuml.org>
17. Yin, X., Javier, A., Machida, F., Andrade, E., Trivedi, K.: Availability Modeling and Analysis for Data Backup and Restore Operations. In: SRDS (2012)

A Controlled Experiment on Component Fault Trees

Jessica Jung¹, Andreas Jedlitschka¹, Kai Höfig², Dominik Domis^{3,*}, and Martin Hiller⁴

¹ Fraunhofer IESE, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{name.surname}@iese.fraunhofer.de

² University of Kaiserslautern, 67653 Kaiserslautern, Germany
hoefig@cs.tu-kl.de

³ ABB Corporate Research Germany, Industrial Software Systems Program, 68526 Ladenburg
Dominik.Domis@de.abb.com

⁴ Cassidian, 89077 Ulm, Germany

Martin.Hiller@cassidian.com

Abstract. In safety analysis for safety-critical embedded systems, methods such as FMEA and fault trees (FT) are strongly established in practice. However, the current shift towards model-based development has resulted in various new safety analysis methods, such as Component Integrated Fault Trees (CFT). Industry demands to know the benefits of these new methods. To compare CFT to FT, we conducted a controlled experiment in which 18 participants from industry and academia had to apply each method to safety modeling tasks from the avionics domain. Although the analysis of the solutions showed that the use of CFT did not yield a significantly different number of correct or incorrect solutions, the participants subjectively rated the modeling capacities of CFT significantly higher in terms of model consistency, clarity, and maintainability. The results are promising for the potential of CFT as a model-based approach.

1 Introduction

Fault Trees (FT) [1] are widely used in industry to calculate hazard occurrence probabilities in the certification of avionic systems according to ARP 4754 [2] and DO-178C [3]. This is done by analyzing the propagation of faults through a system, identifying the causes (events) of the hazards, and calculating the probability of the hazards from the probabilities of the basic events. With the advent of model-based development languages such as SysML [4], which were introduced to tame the complexity of, e.g., avionic systems, the question arose of whether and how safety analysis can also benefit from the model-based paradigm. A whole series of new analysis techniques and adaptations of existing ones have been proposed, such as semantic transformation [5], fault injection [6], and failure logic modeling [7] approaches. However, the effectiveness and the advantages of most approaches have only been argued or validated in single case studies. None of them is widely used in industry; not least due to the fact that they lack thorough empirical evaluation. To provide this evidence for Component Integrated Fault Trees (CFT) [8][9], namely a Failure Logic

* At the time of the experiment, the author was with the University of Kaiserslautern, Germany.

Modeling approach, we performed a controlled experiment with participants from academia and practitioners from Cassidian, an EADS company. The experiment compared CFTs with FTs in analyzing an avionic system modeled as SysML block diagrams [4]. The participants solved four analysis tasks with each method and gave feedback on both approaches by means of a questionnaire.

This paper is structured as follows. Chapter 2 presents related work, i.e., experiments on safety-related aspects. In Chapter 3, we describe our experiment and analyze the results. In Chapter 4, we discuss the threats to validity and present lessons learned in Chapter 5. In Chapter 6, we summarize the paper and conclude our work.

2 Related Work

Although studies have been published on safety analysis approaches, controlled experiments are rarely reported. Stålhane and colleagues published a set of controlled experiments with students comparing different approaches in the area of security and safety with respect to the subjects' performance, i.e., the number of identified hazards. They compared misuse cases (MUC) with FMEA [10], MUC with textual use cases (TUC) [11], and system sequence diagrams (SSD) with TUC [12]. Like FT, FMEA can be used for failure propagation analyses, but it is used here for hazard identification. Learning from those experiments, the assessment of participants' perception is an appropriate complement to objective metrics such as number of correct solutions.

Briand et al. [13] investigated the impact of SysML design slices on inspectors' decision correctness and effort, which is important for safety certification, in a controlled experiment with 20 graduate students. The results show a significant decrease in effort and an increase in the correctness and level of certainty of the decisions. Pai and Dugan [14] empirically evaluated their Bayesian network (BN) model, relating object-oriented software metrics to software fault content and fault proneness, by using a public domain data set from a software subsystem.

In conclusion, empirical work and some controlled experiments have been published in the safety domain, but their focus is on other aspects such as hazard identification or security and not on model-based failure propagation analyses like the controlled experiment described in this paper for fault trees.

3 Experiment

FT (Fig.1a) and CFT (Fig.1b) model Boolean formulas as trees with top events such as hazards as roots, Boolean gates such as And and Or as nodes, and basic failure modes such as component failures as leaves. CFT and FT provide the same qualitative and quantitative analyses, such as calculating hazard occurrence probabilities and minimal cut sets. However, FT show one separated tree for each top event. In contrast, CFT [8][9] modularize many trees into fault tree components with many input and output failure modes according to the structure of a component model such as SysML block diagrams or Matlab/Simulink subsystems. CFT are nested as components. In addition, a CFT is formally as well as graphically integrated with the model

of a component by corresponding associations. The same is true for the in-/output failure modes of a CFT and the input and output interfaces (ports) of its associated component. The CFT representation is thus closer to the model of a system and its components, and should facilitate both traceability between the system and the safety model and safety analysis. In the experiment, the purpose of our research was to check whether the quality of the modeling results obtained from the application of CFT differs from those obtained using FT, and whether the participants would favor CFT over FT. Thus, we specified research questions RQ1 and RQ2:

RQ1: *Will the application of the CFT yield the same quality of the resulting safety model compared to a model built with FT?* RQ1 relates to the performance of the techniques and was investigated by analyzing the quality of the participants' results in modeling tasks performed with the two techniques. We define quality as correctness with three instances: *number of correct solutions* (all task solutions that are completely correct or follow the correct logic), *number of incorrect solutions* (all attempts of task solutions that do not follow the correct logic, use the wrong system element, or are incomplete), and *number of missing solutions* (all tasks that were not worked on by the participant, meaning that no marks or attempts to solve the problem are visible). For this comparison, we stated the following hypothesis¹:

- H1: When using CFT, participants will produce results with a different level of quality compared to when they use FT.
- H1.1: # correct solutions CFT \neq # correct solutions FT
- H1.2: # incorrect solutions CFT \neq # incorrect solutions FT
- H1.3: # missing solutions CFT \neq # missing solutions FT

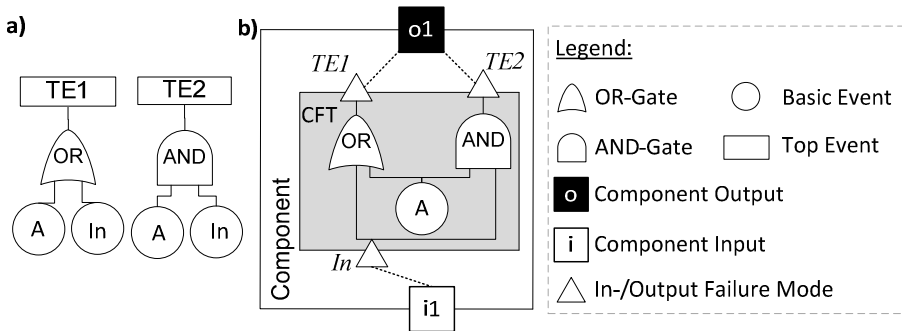


Fig. 1. a) Two FT and b) corresponding CFT integrated with its component

RQ2: *Is CFT perceived different than FT with regard to consistency, clarity, and maintainability?* To answer RQ2, the participants were asked to provide their opinion in a questionnaire after each modeling task. Subjective perception focuses on three important fault tree qualities: *consistency* (the system design model and the fault tree describe the same system), *clarity* (the graphical presentation of the system supports

¹ Due to space limitations, we omit stating the null hypothesis for all presented hypotheses.

keeping an overview of the system), and *maintainability* (changes in the fault trees due to updated system models are easy to administer). Ratings were given on a scale from one (lowest) to five (highest). We stated three hypotheses:

- H2: When using CFT, consistency between system description and safety analysis model is perceived differently than when using FT². (H2: $\mu_{\text{CFT}} \neq \mu_{\text{FT}}$)
- H3: When using CFT, the clarity of safety analysis models developed with CFT is perceived differently than when using FT. (H3: $\mu_{\text{CFT}} \neq \mu_{\text{FT}}$)
- H4: When using CFT, the maintainability of safety analysis models developed with CFT is perceived differently than when using FT. (H4: $\mu_{\text{CFT}} \neq \mu_{\text{FT}}$)

To test the hypotheses, we used the McNemar Test for H1 and the Wilcoxon signed-rank test (a non-parametric test used for dependent samples) for H2-H4 on a significance level of 0.95 ($\alpha = 0.05$).

The study was designed as a three-hour experiment under laboratory conditions including a 40-minute tutorial session about SysML, CFT, and FT before the actual study started. To provide deep cognitive processing of the two methods, we designed four tasks according to typical activities performed by safety engineers. In task 1, the participants were asked to include two missing elements (“basic events”) in the given fault tree and perform the corresponding changes. In task 2, a new, additional functionality of the system was to be included in the given error analysis model. For the solution of task 3, the participants had to implement an additional component in the system and transfer all necessary changes in the corresponding fault trees or component fault trees, respectively. In task 4, an additional analog input component was added to an already existing component in the model. The four tasks were presented in two versions to all participants: version “a” for CFT and version “b” for FT (for reasons of comparability, the tasks for applying either the CFT or the FT method were identical). To avoid ordering effects such as practice, learning effects, or fatigue, the sequence of the eight tasks was randomized for each participant, resulting in a different task order for each individual (within-subject design). Before proceeding with the next task, the participants had to complete a questionnaire consisting of 11 items (=statements) to assess the subjective rating of the perceived consistency (4 items), clarity (4 items), and maintainability (3 items) on an ordinal 5-point Likert scale (1=strongly disagree to 5=strongly agree) for each task. Depending on the task type (a or b), the items contained either the term “CFT” or “FT”.

Although FT and CFT are usually applied with PC software tools, the study was implemented for paper & pencil use to avoid effects caused by the usability of a tool. The participants were instructed to draw missing elements directly in the diagrams provided and cross out non-essential elements. Blank pages were provided to redraw elements. In addition, the system model was provided as a reference document.

After a pilot with several colleagues, the experiment was conducted with 7 academic staff members from the computer science department of the University of

² μ symbolizes the mean value of a variable.

Kaiserslautern (TU KL) and 11 practitioners from Cassidian (sample $n=18$). The academic staff members are PhD students with experience in FT and CFT. All Cassidian employees (8 aerospace engineers, 2 mathematicians, 1 physicist) have several years of experience in safety and FT. Due to company restrictions, no further information was collected. Participation was on a voluntary basis without any compensation.

To answer RQ1, a total of 144 task solutions (18 participants*4 tasks*2 methods) were analyzed by applying a coding scheme based on the definition of correct, incorrect, and missing solutions. The total numbers for the three categories for CFT and FT are shown in **Table 1**. FT has a higher number of correct solutions but also a slightly higher number of incorrect solutions, whereas CFT has a higher number of missing task solutions. We used the McNemar Test (two-tailed; $\alpha = 0.05$) to test the differences between CFT and FT. Regarding the number of correct solutions (H1.1) as well as the number of incorrect solutions (H1.2), there is no significant difference between CFT and FT ($p = 0.126$; $p = 0.688$). For both we retain the null hypothesis: by using CFT, the participants did not produce significantly more or fewer correct as well as incorrect solutions. Concerning the number of missing task solutions (H1.3), the null hypothesis can be rejected, meaning that CFT and FT differ significantly in terms of the number of missing task solutions ($p = 0.006$; $p < \alpha$).

Table 1. Number of correct, incorrect or missing solutions

<i>Method used</i>	<i># correct solutions</i>	<i># incorrect solutions</i>	<i># missing task solutions</i>
CFT	52	6	14
FT	59	8	5
<i>p</i>	0.126	0.688	0.012

For RQ2 we analyzed the participants' subjective ratings about the two methods with respect to the three qualities consistency, clarity, and maintainability. Based on the results of the statistical reliability analysis for the measurement instrument, all items for one quality were merged (sum of values divided by number of items) into one value representing the participants' perception of this quality³. The descriptive results of the three computed values for the four tasks are shown in Table 2. Although mean values are not used for statistical inference analyses here, it can be seen that CFT received higher assessments for all four tasks with respect to all qualities. Only for task 4, the assessment of maintainability does not show any difference.

Regarding H2-H4, we compared CFT and FT across tasks 1-4 combined with the two-tailed Wilcoxon signed-rank test ($\alpha = 0.05$). The participants rated the consistency (H2) of CFT significantly higher (Mdn = 4) than that of FT (Mdn = 4), $z = -2.02$, $p = 0.04$, $r = -0.34$. The null hypothesis is rejected because $p < \alpha$. Regarding variable clarity (H3), CFT was also rated significantly higher (Mdn = 4) than FT (Mdn = 3), $z = -1.98$, $p = 0.04$, $r = -0.33$. Again, the null hypothesis is rejected. In terms of maintainability (H4), the participants assessed CFT significantly higher (Mdn = 4) than FT (Mdn = 3.5), $z = -2.15$, $p = 0.03$, $r = -0.36$. Here, too, the null hypothesis is rejected.

³ The statistical reliability analysis revealed good to excellent reliability of the developed items (Cronbach's $\alpha > 0.84$).

Table 2. Descriptive results of the subjective ratings

		<i>Task 1</i>		<i>Task 2</i>		<i>Task 3</i>		<i>Task 4</i>		<i>Task 1-4 combined</i>	
		<i>CFT</i>	<i>FT</i>	<i>CFT</i>	<i>FT</i>	<i>CFT</i>	<i>FT</i>	<i>CFT</i>	<i>FT</i>	<i>CFT</i>	<i>FT</i>
Consistency	Median	4	4	4	3	4	4	4	4	4	4
	Mean	3.75	3.51	3.59	2.98	3.85	3.60	3.79	3.57	3.75	3.42
	SD	0.93	0.99	0.96	1.03	0.99	0.98	0.99	1.02	0.97	1.00
Clarity	Median	4	3.5	4	3	4	3.5	4	3	4	3
	Mean	3.53	3.12	3.60	2.92	3.79	3.32	3.81	3.32	3.68	3.17
	SD	1.09	1.11	1.16	1.05	1.13	0.94	0.93	1.02	1.08	1.03
Maintainability	Median	4	3	3	3	4	4	4	4	4	3.5
	Mean	3.63	3.29	3.53	3.04	3.92	3.55	3.71	3.73	3.70	3.40
	SD	1.15	1.03	0.99	1.00	1.07	0.96	1.03	0.96	1.06	0.99

4 Threats to Validity

Concerning the internal validity of the study, maturation effects (e.g., fatigue) on the assessment of the two methods may be excluded because of the randomized order of the tasks. Standardized instructions were used to minimize influences caused by experimenter expectancies towards the assessment of the two methods. Furthermore, the participants were told to base their ratings on the experiences of the tasks and not on experiences outside the study. An “independent” researcher led the experiment.

With regard to the external validity of the study, we are aware that due to the example used and the sample selected, the results are only valid for the avionics domain. The tasks, although small in size and complexity, were developed together with practitioners to ensure realism. Moreover, the laboratory conditions as well as the use of paper & pencil are limiting factors. Some information regarding the participants’ experience (e.g., company affiliation time) was not allowed to be collected.

In terms of conclusion validity, appropriate statistical test procedures were used. One possible threat to construct validity is reactivity to the experimental situation. We reduced experimenter interactions with the participants by including all relevant instructions and materials (tasks and corresponding questionnaires) in one package (“test booklet”). The test booklet was self-explanatory so that after the tutorial ended, no interaction between experimenter and participant was necessary. Furthermore, anonymity and confidentiality were assured. The study procedure, material, and instrumentation were tested in three pilot studies, yielding improvements concerning understandability, task descriptions, and procedure.

5 Lessons Learned

Based on the results and taking into consideration the threats to validity, we conclude that our participants rated CFT subjectively better than FT, although the use of CFT yielded neither significantly more correct nor more incorrect solutions. One problem in comparing the task results for CFT and FT is the large number of missing solutions in the CFT condition (14 out of 72) compared to the FT one (5 out of 72).

One possible explanation is that some participants already had more experience with FT, whereas CFT was unknown before. They may have felt uncertain in applying CFT

and preferred not to provide a solution instead of making a mistake. Not solving an experimental task means in practice that the task is still open and more resources or information must be incorporated to solve the task. This procedure is safe and avoids wrong solutions that might negatively affect system safety. This assumption is supported by the fact that the number of correct solutions for CFT did not differ significantly from the number of correct solutions for FT and the results of the other hypotheses. Taking into consideration the short duration of the tutorial (approx. 40 min), we interpret this result in favor of CFT and hypothesize that with a more detailed tutorial and hands-on experience, even better results for CFT could have been achieved. However, it should be investigated in future studies whether this assumption is true. If in another context engineers were to produce wrong instead of missing task solutions, this might affect system safety negatively and must be avoided.

The similarity of the graphical representation of the system design model and the CFT-based fault tree is reflected in the subjective ratings. Regarding the three main qualities (consistency, clarity, and maintainability), CFT was subjectively assessed as being significantly better than FT. We believe that software engineers will prefer software development methodologies, for safety analysis as well as for software in general, which are intuitively closer to people's cognitive skills [15].

Based on the results, we conclude that CFT, being more similar to model-based design approaches than FT, can be beneficial for employees with little or no experience in FT. However, if employees have experience in FT, the implementation effort for introducing CFT has to be balanced against a possible positive outcome regarding employees' positive subjective assessment and the quality of the safety analysis.

6 Summary and Conclusion

In this paper, we have presented a controlled experiment comparing CFT, which adhere to the graphical logic of the system design model, to standard FT, which follow a completely different graphical approach. The experiment consisted of the analysis of a real avionic system, performed by PhD students and practitioners with many years of work experience from Cassidian, an EADS company. Our research question addressed two aspects: 1) the quality (completeness and correctness) of modeling task solutions and 2) the participants' subjective perception with regard to the clarity, consistency, and maintainability of the methods. We developed an empirical study design in a laboratory setting with realistic tasks and a realistic system model from the avionics domain. The results of the study showed that when applying CFT, the participants did not produce significantly more or fewer correct and incorrect solutions compared to FT. A larger number of missing task solutions for CFT might be attributed to uncertainty in applying the CFT method compared to the well-known FT. In terms of subjective assessment of the two methods, CFT was rated higher regarding consistency, clarity, and maintainability: The participants favored CFT over FT. Because this subjective perception is based on experiences, it might indicate that CFT could also show advantages with regard to the quality of analysis results for larger and more complex systems than those we were able to analyze in the experiment. To analyze this question, a long-term study would be

recommended, which is our aim for future work in addition to improving the method and the experiments for such methods.

Acknowledgments. We would like to thank the participants of our studies. Parts of this work were funded by the German Ministry of Education and Research under grant number 01IS0804 and 01IS12005E in the context of the project SPES2020 and SPES2020_XTCore.

References

1. International Electrotechnical Commission, Fault tree analysis (FTA), IEC 61025 ed2.0 (December 13, 2006)
2. SAE International, Guidelines for Development of Civil Aircraft and Systems. ARP4754A (2010)
3. Radio Technical Commission for Aeronautics Software, Considerations in Airborne Systems and Equipment Certification. DO-178C (2012)
4. Object Management Group: OMG Systems Modeling Language, <http://www.omg.sysml.org/> (last visited March 9, 2013)
5. de Miguel, M.A., Briones, J.F., Silva, J.P., Alonso, A.: Integration of safety analysis in model-driven software development. *IET Software* 2(3), 260–280 (2008)
6. Damm, W., Votintseva, A., Metzner, A., Josko, B., Peikenkamp, T., Böde, E.: Boosting Re-use of Embedded Automotive Applications Through Rich Components. In: Elsevier's Electronic Notes in Theoretical Computer Science. Elsevier Science B.V. (2005)
7. Papadopoulos, Y., McDermid, J.A.: Hierarchically Performed Hazard Origin and Propagation Studies. *Computer Safety, Reliability and Security* (1999)
8. Domis, D., Hoefig, K., Trapp, M.: A Consistency Check Algorithm for Component-based Refinements of Fault Trees. In: Proc. 21st IEEE Intern. Symposium on Software Reliability Engineering (ISSRE), San Jose CA, USA, pp. 171–180 (2010)
9. Adler, R., Domis, D., Hoefig, K., Kemmann, S., Kuhn, T., Schwinn, J., Trapp, M.: Integration of Component Fault Trees into the UML. Non-functional System Properties in Domain Specific Modeling Languages. In (NFPinDSML 2010), Workshop at ACM/IEEE 13th Intern. Conf. on Model Driven Engineering Languages and Systems, Oslo, Norway (2010)
10. Stålhane, T., Sindre, G.: A Comparison of Two Approaches to Safety Analysis Based on Use Cases. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 423–437. Springer, Heidelberg (2007)
11. Stålhane, T., Sindre, G.: Safety Hazard Identification by Misuse Cases: Experimental Comparison of Text and Diagrams. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 721–735. Springer, Heidelberg (2008)
12. Stålhane, T., Sindre, G., du Bousquet, L.: Comparing safety analysis based on sequence diagrams and textual use cases. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 165–179. Springer, Heidelberg (2010)
13. Briand, L., Falessi, D., Nejati, S., Sabetzadeh, M., Yue, T.: Traceability and SysML Design Slices to Support Safety Inspections: A Controlled Experiment. Technical Report, Simula Research Laboratory (August. 2010)
14. Pai, G.J., Dugan, J.B.: Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods. *IEEE Trans. Software Eng.* 33(10), 675–686 (2007)
15. Paivio, A.: *Mind and Its Evolution: A Dual Coding Theoretical Approach*. Lawrence Erlbaum, Mahwah (2006)

DFTCALC: A Tool for Efficient Fault Tree Analysis

Florian Arnold, Axel Belinfante, Freark Van der Berg,
Dennis Guck, and Mariëlle Stoeltinga

Department of Computer Science, University of Twente, The Netherlands
{f.arnold,d.guck,a.f.e.belinfante,m.i.a.stoeltinga}@utwente.nl,
f.i.vanderberg@student.utwente.nl

Abstract. Effective risk management is a key to ensure that our nuclear power plants, medical equipment, and power grids are dependable; and it is often required by law. Fault Tree Analysis (FTA) is a widely used methodology here, computing important dependability measures like system reliability. This paper presents DFTCALC, a powerful tool for FTA, providing (1) efficient fault tree modelling via compact representations; (2) effective analysis, allowing a wide range of dependability properties to be analysed (3) efficient analysis, via state-of-the-art stochastic techniques; and (4) a flexible and extensible framework, where gates can easily be changed or added. Technically, DFTCALC is realised via stochastic model checking, an innovative technique offering a wide plethora of powerful analysis techniques, including aggressive compression techniques to keep the underlying state space small.

1 Introduction

Risk analysis is a key feature in reliability engineering: in order to design and build medical devices, smart grids, and internet shops that meet the required dependability standards, we need to assess at design time how dependable these systems are, and take appropriate measures if they are not dependable enough.

Fault Trees. Fault tree analysis (FTA) [19] is a graphical technique that is often used in industry. Fault trees (FTs) model how component failures lead to system failures: the leaves of a FT are basic events (BEs) that represent component failures; the other nodes express how failures propagate through the system via AND and OR gates. Discrete time FTs equip each BE with a probability p , representing the probability that the component fails within a certain discrete time interval. We consider continuous FTs. Here, each BE is equipped with a probability distribution f showing how the failure behaviour evolves over time, i.e. $F(t)$ represents the probability that the BE is still running at time point t . The root of the tree, called the top-level event, represents a system failure. FTA typically computes for a given FT the *system reliability*, i.e. the probability that the system has not failed within a given mission time T , the *mean time to failure* (MTTF), i.e. the expected time of a failure to occur, and the *availability*, i.e. the time that the system is up in the long run.

Dynamic Fault Trees (DFTs) extend standard (or static) fault trees with a number of intuitive gates. These gates facilitate the modelling of often recurring concepts in reliability engineering: spare management, functional dependencies, and order-dependent behaviour.

DFTCalc. DFTCALC is a powerful tool for modelling and analysis of DFTs. It can efficiently model DFTs and provides means to compute various dependability metrics, given BEs whose failure probabilities are given by exponential and phase type distributions. The major innovation of DFTCALC is the deployment of stochastic model checking (SMC) techniques [4]: SMC is an innovative technique to systematically explore the state space of a stochastic system. SMC provides a wide plethora of powerful analysis techniques, with fully-fledged tool support. By deploying SMC, DFTCALC can handle DFTs with BEs that are statistically dependent; in fact, the FDEP gate has specifically been designed to model interdependent events. Repairs, however, have not yet been included.

The main problem in time-dependent reliability analysis is its complexity: The state space of models of real systems can grow arbitrarily large [15] and, thus, highly efficient techniques are required to yield results in a feasible time. Furthermore, an accurate modelling of all dependencies in these inherently complex systems requires an ever growing diversity of new gates. DFTCALC constitutes an architectural framework that addresses both challenges.

Related Work. A wide range of FTA methods exists: Classically, one obtains the minimal cut sets in the FT [5]. This enables to order components based on their structural importance. Further, with additional information one can compute the system reliability. A popular technique is to exploit Bayesian networks, which are useful both in discrete time [9] and in continuous time [8]. Our approach focuses on continuous timed systems, with currently no maintenance. Therefore, we will translate DFTs into continuous time Markov chains (CTMCs) and use state of the art techniques as described in [2,3]. This allows us to compute reliability measures by use of efficient techniques for transient analysis of CTMCs.

A wide number of commercial and academic tools for static fault tree analysis are available. Some are merely drawing tools, while others provide probabilistic analysis, like the popular FaultTree+ package from Isograph [14]. Dynamic FTA is supported by tools like Windchill [18], NASA's Galileo/ASSAP software [11], and the simulation tool DFTSim [10]. A first implementation of DFT analysis using I/O-IMCs was realized in Coral [7], the predecessor of DFTCALC.

Organisation of the Paper. Section 2 presents DFTCALC's modelling and analysis capabilities and Section 3 the architecture and internal structure. In Section 4 we provide experimental results and Section 5 concludes the paper. Due to space constraints, we refer to [1] for more details of our main results and case studies.

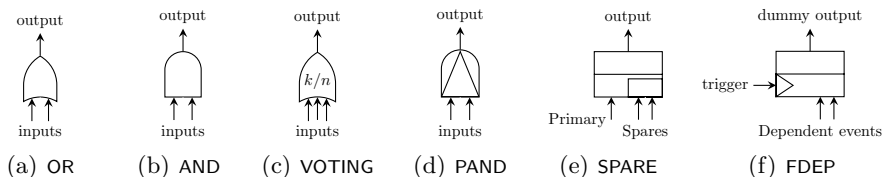


Fig. 1. Dynamic fault tree gates

2 DFTCALC: Modelling and Analysis

DFT Modelling. Dynamic fault trees (DFTs) model the failure propagation in complex systems. The leaves of a DFT are labeled with *basic events* and the non-leaves with *gates*. The root is called *top-level event*.

Basic Events. A basic event (BE) represents the failure behaviour of a basic system component, and can be in three different modes: *dormant*, *active* and *failed*. The component is in dormant mode, if it is not in use. In this mode, the failure rate of a BE is decreased by a *dormancy factor* $\alpha \in [0, 1]$. In case $\alpha = 0$ the BE cannot fail (cold BE) and in case $\alpha = 1$ the failure rate is the same as in active mode (warm BE). The component is in active mode, when it is in use. If the component breaks down, it is in failed mode.

Gates. A gate expresses how component failures induce a system failure. Gates consist of one or more inputs, and one output. Fig. 1 depicts the DFT gates.

- (a) The OR gate fails when at least one input fails.
- (b) The AND gate fails when all of its inputs fail.
- (c) The VOTING gate fails when at least k out of n inputs fail.
- (d) The PAND gate fails when all of its inputs fail from left to right.
- (e) The SPARE gate consists of a *primary* input and one or more *spare* inputs. At system start, the primary is active and the spares are in dormant mode. When the primary input fails, one of the spare inputs is activated and replaces the primary. If no more spares are available, the SPARE gate fails. Note that a spare component can be shared among several spare gates.
- (f) The FDEP (functional dependency) gate consists of one *trigger* event and several *dependent events*. When the trigger event occurs, all dependent events fail. The FDEP has a "dummy" output, which is represented by a dotted line and ignored in calculations.

Example 1. Fig. 2 depicts a DFT representing a cardiac assist system (CAS) [9] consisting of three subsystems: the CPU, the motor and pump units. If either one of these subsystems fails, then the entire CAS fails, as modelled by the top level OR gate. The CPU unit consists of a primary (P) and a backup (B) CPU, as indicated by the SPARE gate. The primary and backup CPU are subject to a common cause failure, modelled by the CPU FDEP gate: if either the crossbar

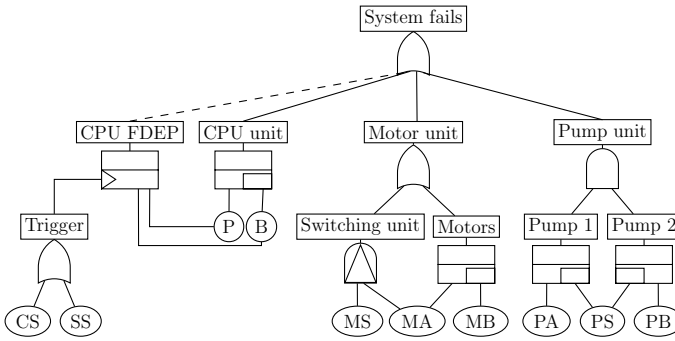


Fig. 2. The cardiac assist system DFT

switch (CS) or the system supervisor (SS) fails, the primary and backup CPU become unavailable. The motor unit consists of a primary (MA) and a backup (MB) motor. If the primary fails, the motor switching component (MS) will turn on the backup motor. Because of the PAND gate the failure of the switching component can then be ignored. Finally, the pump unit consists of two pumps (PA and PB), which share a common cold spare (PS).

DFT Analysis. DFTCALC can compute a number of different reliability metrics, namely all metrics that can be expressed as reachability properties in the logic CSL. This includes properties such as: (1) *Timed-Reliability*: the probability that the system fails until a given time point T or in a given interval $[T, T']$; (2) *Mean time to failure*: the expected time to a system failure; (3) *Reliability*: the probability that the system fails in the long-run. In case of non-determinism, we calculate the minimum and maximum values for the above metrics. Each of these properties can either be evaluated from the initial state (i.e. the system is fully functional), or by *setting evidence* (i.e. certain components have failed already).

DFTCALC fruitfully exploits the technique of *compositional aggregation*, see Fig. 6. Whereas traditional FTA methods translate a DFT into a large and monolithic CTMC, we do this in a stepwise fashion: First, DFTCALC translates each element (i.e., gate or BE) into an input-output interactive Markov chain (I/O-IMC), implementing the methodology from [6,7]. Then, we obtain the underlying CTMC by composing all I/O-IMCs. We compose these I/O-IMCs one-by-one, and employ aggressive state space compression technique in each step, to keep the state space minimal. This compositional approach has four major advantages:

- *Increased modelling power.* Compared to earlier DFT tools, DFTCALC’s input language is more powerful and imposes fewer syntactic restrictions: DFTCALC allows any DFT to be a spare component or a trigger, and not only a BE, as in [16]. This is a big advantage in practice, since spare components and triggers are often complete subsystems.

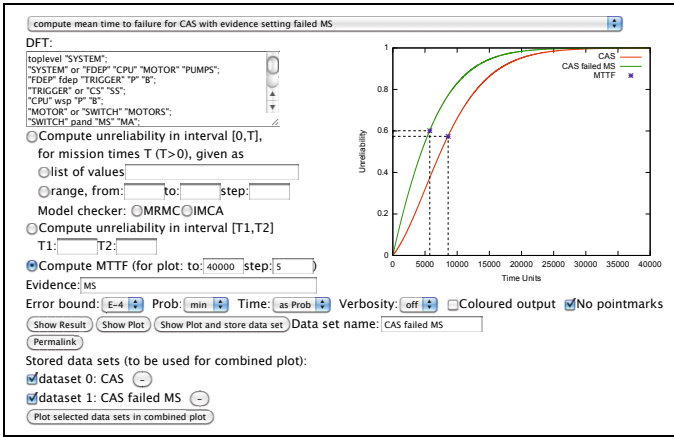


Fig. 3. DFTCALC web-tool interface

- *Increased analytical power.* SMC enables DFTCALC to analyse a wide range of dependability metrics, namely those expressed in a large subset of the logic CSL. Also, as argued in [6], certain DFTs give rise to non-determinism. If so, the I/O-IMC leads to a continuous time Markov decision process (CTMDP).
- *Efficiency.* The compositional aggregation technique leads to significant speed ups of several orders of magnitude.
- *Flexibility.* The compositional aggregation approach makes the framework very extendable. In order to change the behaviour of a gate or even add new gate types, we only need to provide the underlying I/O-IMC model.

Web Interface. DFTCALC can be used by downloading a stand-alone version, and via a web interface. Both are accessible at <http://fmt.cs.utwente.nl/tools/dftcalc/>. DFTCALC is open source, but requires a license for CADP, which is free for academic institutions. The web interface extends the downloadable version with a GUI as well as the plot function and is shown in Fig. 3. It allows the user to (1) input DFT models via a text screen, the topmost box in Fig. 3; (2) select the dependability metrics. This can be (a) the reliability for one or more mission times x , or (b) the probability on a system failure during an interval $[T1, T2]$, or (c) the mean time to failure; (3) set various options: which model checker to use; the error bound, the level of verbosity, and whether to color output. The results can be given either by numbers, via the button *show*

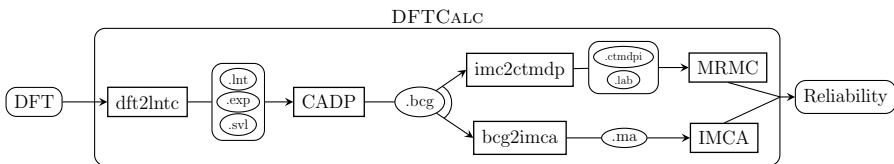


Fig. 4. The DFTCALC tool-chain

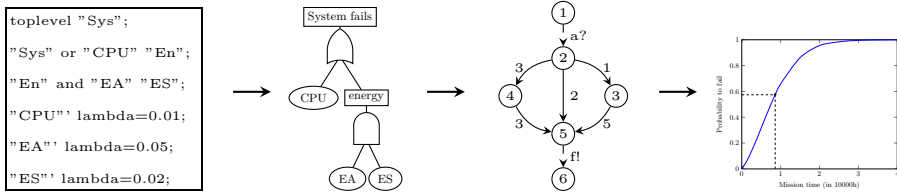


Fig. 5. Graphical overview of the processing steps in DFTCALC

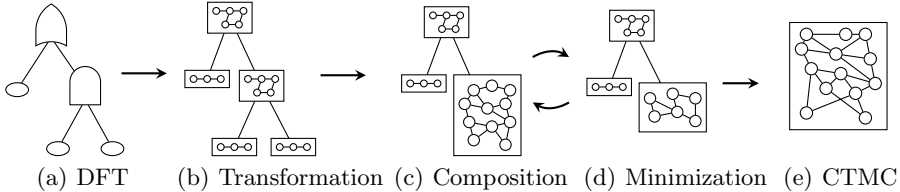


Fig. 6. Graphical overview of the compositional aggregation of DFT models

result, or as a plot, via the button *plot result*. The input and configuration of the web interface can be saved via the button *permalink*.

3 DFTCALC’s Internal Structure

Architecture. DFTCALC combines dedicated code and state-of-the-art model checkers. The architecture is displayed in Fig. 4 and the processing steps in Fig. 5: First, `dft2lntc` translates a DFT in Galileo format into `.lnt` format, a process calculus enriched with data that is input to CADP. Technically, this step transforms each DFT element into an I/O-IMC representing the element’s behaviour. Additionally, a `.exp` file is generated that defines the interaction between components. The clear distinction between local component and global system information together with the compositional semantics of I/O-IMCs makes DFTCALC highly flexible: New components can be added or existing components adapted by specifying their behaviour in `.lnt` format and adding them to the tool’s library. In the next step, the CADP tool set [12] uses the compositional aggregation method to generate the state space of the system, which is a I/O-IMC representation of the whole DFT. The output of CADP is a `.bcg` file. This format is translated either into a `.ctmdpi` file, which is input to the Markov Reward Model Checker MRMC [15], or into an `.ma` file, which is the input of the Interactive Markov Chain Analyzer IMCA [13]. Finally, the requested dependability metrics are computed.

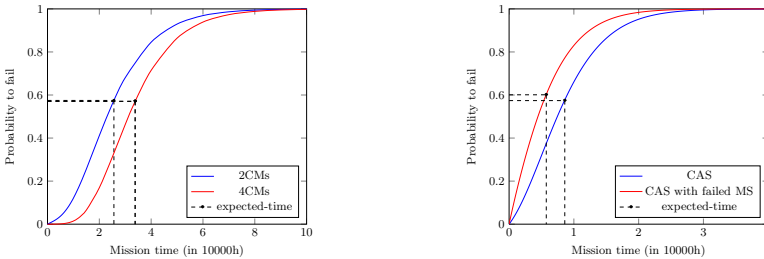
Compositional Aggregation. Compositional aggregation of I/O-IMCs lies at the heart of DFTCALC. As depicted in Fig. 6, after transforming each DFT element into an I/O-IMC, we iteratively compose the obtained I/O-IMCs: We take two I/O-IMCs, compose them, hide all action labels that are no longer

needed for synchronisation, and then minimise the composition via bisimulation minimisation. This process continues until a single I/O-IMC remains. The order of the aggregation process heavily influences the number of states in the obtained I/O-IMC, and is determined by a smart heuristic. Compositional aggregation yields reductions up to several orders of magnitude [7].

4 Case Studies

We show the applicability of DFTCALC by three case studies: a multiprocessor computing system (MCS) [17,7] which consists of two computing modules (CMs), a bus, a power supply and a spare memory module; the cardiac assist system (CAS) [9,8] from Fig. 2; and a fault-tolerant parallel processor (FTPP) [7] of a redundant computer system consisting of four groups of n processors. The MCS and CAS models were originally developed for discrete time models [17,9], but were analyzed, as we do, for continuous time models in [7,8].

All our experiments were conducted on a single core of a 2.7 GHz Intel Core2Duo processor with 2GB RAM running on Linux. Fig. 7 presents the increasing failure probability over time as well as the expected failure time. Table 1 shows the scalability. We compare Coral and DFTCALC: Since DFTCALC is up to three times faster than Coral it also outperforms earlier tools like Galileo [7].



(a) Failure probability of the MCS over time. (b) Failure probability of the CAS over time.

Fig. 7. Reliability plots for the case studies

Table 1. Results of the case studies

Model	Tool	Time (s)	P(fail)	States	Transitions	Speedup
MCS 2CMs, t=10000	Coral	131.492	0.998963	18	55	1
	DFTCALC	55.395	0.998963	18	55	2.37371
MCS 4CMs, t=10000	Coral	339.752	0.997927	151	992	1
	DFTCALC	201.461	0.997927	151	992	1.68644
CAS, t=10000	Coral	135.155	0.0460314	16	50	1
	DFTCALC	51.267	0.0460314	16	50	2.64794
FTPP-4, t=1	Coral	491.114	0.0192186	142	923	1
	DFTCALC	234.905	0.0192186	72	386	2.09069
FTPP-5, t=1	Coral	730.761	0.0030616	2167	27438	1
	DFTCALC	603.630	0.0030616	400	3369	1.21061

5 Conclusion

We have presented an efficient tool chain which allows to model and analyse DFTs with a number of prominent dependability metrics. The flexible architecture of DFTCALC exploits state-of-the-art techniques to compose, compress and analyse DFTs, and is easily extendable. We have conducted several case studies demonstrating DFTCALC's high performance in the analysis of DFTs.

As future work, we aim to include cost structures and repairable basic events. Moreover, we will use DFTCALC's flexible architecture to implement additional gates to broaden DFTCALC to other formalisms like attack trees.

Acknowledgements. This research has been partially funded by the NWO under the project ArRangeer (12238), and by the DFG/NWO bilateral project ROCKS (DN 63-257) and by the EU FP7 under the project TRESPASS (318003).

References

1. Arnold, F., Belinfante, A., Van der Berg, F., Guck, D., Stoelinga, M.: Dftcalc: a tool for efficient fault tree analysis (extended version). Technical Report TR-CTIT-13-13, CTIT, University of Twente, Enschede (June 2013)
2. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. *IEEE TSE* 29(6), 524–541 (2003), doi:10.1109/TSE.2003.1205180
3. Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. *Theoretical Computer Science* 345(1), 2–26 (2005)
4. Baier, C., Katoen, J.-P.: *Principles of Model Checking*. MIT Press (2008)
5. Barlow, R.E., Proschan, F.: *Statistical theory of reliability and life testing: probability models*. Holt, Rinehart and Winston (1975)
6. Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic fault tree analysis using Input/Output interactive Markov chains. In: *DSN*, pp. 708–717 (2007)
7. Boudali, H., Crouzen, P., Stoelinga, M.: A rigorous, compositional, and extensible framework for dynamic fault tree analysis. *IEEE TDSC* 7, 128–143 (2010)
8. Boudali, H., Dugan, J.: A continuous-time bayesian network reliability modeling and analysis framework. *IEEE Transactions on Reliability* 55(1), 86–97 (2006)
9. Boudali, H., Dugan, J.B.: A Bayesian network reliability modeling and analysis framework. *IEEE Transactions on Reliability* 55, 86–97 (2005)
10. Boudali, H., Nijmeijer, A.P., Stoelinga, M.: DFTSim: A simulation tool for extended dynamic fault trees. In: *ANSS 2009*, p. 31 (2009)
11. Coppit, D., Sullivan, K.: Galileo: A tool built from mass-market applications. In: *International Conference on Software Engineering*, pp. 750–753 (2000)
12. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 1–19 (2012)
13. Guck, D., Han, T., Katoen, J.-P., Neuhäüßer, M.R.: Quantitative timed analysis of interactive Markov chains. In: Goodloe, A.E., Person, S. (eds.) *NFM 2012*. LNCS, vol. 7226, pp. 8–23. Springer, Heidelberg (2012)

14. Isograph. Fault Tree +, www.isograph-software.com/2011/software/
15. Katoen, J.-P., Zapreev, I., Hahn, E.M., Hermanns, H., Jansen, D.: The ins and outs of the probabilistic model checker MRMC. *Perf. Eval.* 68(2), 90–104 (2011)
16. Manian, R., Bechta Dugan, J., Coppit, D., Sullivan, K.: Combining various solution techniques for dynamic fault tree analysis of computer systems. In: *Proc. IEEE Int. High-Assurance Systems Engineering Symposium*, pp. 21–28 (1998)
17. Montani, S., Portinale, L., Bobbio, A., Varesio, M., Codetta-Raiteri, D.: A tool for automatically translating dynamic fault trees into dynamic Bayesian networks. In: *RAMS*, pp. 434–441 (2006)
18. PTC. Windchill FTA, <http://www.ptc.com/product/relex/fault-tree>
19. Veseley, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: *Fault tree handbook*, NUREG-0492. Technical report, NASA (1981)

Author Index

- Alata, Eric 146
Albu-Schäffer, Alin 202
Alexander, Rob 33
Alves, Marcelo 277
Andrade, Ermeson C. 277
Arnold, Florian 293
Arnold, James 33
Ayatollahi, Fatemeh 265
Ayoub, Anaheed 228
- Baraza, J.-Carlos 178
Belder, Rico 202
Belinfante, Axel 293
Birch, John 154
Bishop, Peter 118
Böhm, Petr 253
Botham, John 154
Bouffard, Guillaume 82
Bradshaw, Ben 154
- Cukier, Michel 94
Curzon, Paul 228
- Denney, Ewen 21
Dessiatnikoff, Anthony 146
Deswarte, Yves 146
Domis, Dominik 285
- Ern, Bernhard 241
- Fabre, Jean-Charles 45
- Gashi, Ilir 94
Gil-Tomás, Daniel 178
Gil-Vicente, Pedro-J. 178
Górski, Janusz 8
Gracia-Morán, Joaquín 178
Gruber, Thomas 253
Guck, Dennis 293
- Habli, Ibrahim 154
Haddadin, Sami 202
Higham, Dave 154
Hiller, Martin 285
Höfig, Kai 285
- Iliasov, Alexei 130
Ilić, Dubravka 57
- Jarzębowicz, Aleksander 8
Jedlitschka, Andreas 285
Jesty, Peter 154
Johansson, Roger 265
Jung, Jessica 285
- Karlsson, Johan 265
Kleberger, Pierre 70
- Laibinis, Linas 57
Lamedschwandner, Kurt 253
Lanet, Jean-Louis 82
Latvala, Timo 57
Lauret, Jimmy 45
Lee, Insup 228
Lopatkin, Ilya 130
- Maciel, Paulo 277
Martinie, Célia 216
Masci, Paolo 228
Matos, Rubens 277
Miler, Jakub 8
Monkhouse, Helen 154
- Navarre, David 216
Neubauer, Georg 253
Nguyen, Viet Yen 241
Nicomette, Vincent 146
Noll, Thomas 241
Novak, Thomas 138
Nyberg, Mattias 166
- Olovsson, Tomas 70
- Pai, Ganesh 21
Palanque, Philippe 216
Palin, Robert 154
Parusel, Sven 202
Pasquini, Alberto 216
Povyakalo, Andrey 106
Prokhorova, Yuliya 57
- Ragosta, Martina 216
Rivett, Roger 154

- Romanovsky, Alexander 130
Ruiz-García, Juan-Carlos 178
Rushby, John 1
- Saiz-Adalid, Luis-J. 178
Sangchoolie, Behrooz 265
Schiffel, Ute 190
Silva, Bruno 277
Sobesto, Bertrand 94
Sokolsky, Oleg 228
Stankovic, Vladimir 94
Stoegerer, Christoph 138
Stoelinga, Mariëlle 293
- Strigini, Lorenzo 106
Sujan, Mark Alexander 216
- Thampi, Bhagyalekshmy N. 82
Thimbleby, Harold 228
Törngren, Martin 166
Troubitsyna, Elena 57
- Van der Berg, Freark 293
- Waeselynck, Hélène 45
Weinfurter, Andreas 253
Westman, Jonas 166