Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into
• seminars
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: http://www.gi.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-604-6

"Automotive 2012" is the fifth event in a conference series focusing on safety and security in the field of automotive and other critical software applications. This volume contains contributions from the invited papers and the detailed papers presented during the conference.

E.Plödereder et al. (Hrsg.): Automotive – Safety & Security 2012

210

GI-Edition

Lecture Notes in Informatics

Erhard Plödereder, Peter Dencker, Herbert Klenk, Hubert B. Keller, Silke Spitzer (Hrsg.)

Automotive – Safety & Security 2012

Sicherheit und Zuverlässigkeit für automobile Informationstechnik

14.–15. November 2012
Karlsruhe

Proceedings

Erhard Plödereder, Peter Dencker, Herbert Klenk,
Hubert B. Keller, Silke Spitzer (Hrsg.)

Automotive – Safety & Security 2012

Sicherheit und Zuverlässigkeit für automobile Informationstechnik

Tagung
14.-15. 11. 2012 in Karlsruhe

Gesellschaft für Informatik e. V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-210

ISBN 978-3-88579-604-6
ISSN 1617-5468

Volume Editors
Herbert Klenk
Cassidian
Rechliner Str.
85077 Manching
Email: herbert.klenk@cassidian.com
Erhard Plödereder
Institut für Softwaretechnologie, Universität Stuttgart
Universitätsstr. 38
70569 Stuttgart
Email: ploedere@informatik.uni-stuttgart.de

Series Editorial Board
Heinrich C. Mayr, Alpen-Adria-Universität Klagenfurt, Austria
(Chairman, mayr@ifit.uni-klu.ac.at)
Dieter Fellner, Technische Universität Darmstadt, Germany
Ulrich Flegel, Hochschule für Technik, Stuttgart, Germany
Ulrich Frank, Universität Duisburg-Essen, Germany
Johann-Christoph Freytag, Humboldt-Universität zu Berlin, Germany
Michael Goedicke, Universität Duisburg-Essen, Germany
Ralf Hofestädt, Universität Bielefeld, Germany
Michael Koch, Universität der Bundeswehr München, Germany
Axel Lehmann, Universität der Bundeswehr München, Germany
Peter Sanders, Karlsruher Institut für Technologie (KIT), Germany
Sigrid Schubert, Universität Siegen, Germany
Ingo Timm, Universität Trier, Germany
Karin Vosseberg, Hochule Bremerhaven, Germany
Maria Wimmer, Universität Koblenz-Landau, Germany

Dissertations
Steffen Hölldobler, Technische Universität Dresden, Germany
Seminars
Reinhard Wilhelm, Universität des Saarlandes, Germany
Thematics
Andreas Oberweis, Karlsruher Institut für Technologie (KIT), Germany

© Gesellschaft für Informatik, Bonn 2012
printed by Köllen Druck+Verlag GmbH, Bonn

Vorwort

Die 5. Tagung Automotive – Safety & Security 2012 - Sicherheit und Zuverlässigkeit für automobile Informationstechnik fand am 14. - 15. November 2012 in Karlsruhe im Tagungsbereich der Robert Bosch GmbH statt. Die Tagung adressierte die zentralen Themen Zuverlässigkeit und Sicherheit softwarebasierter Funktionen im Automotive Bereich. Über zwei Tage wurden die neuesten Ideen und konkreten Lösungen für die drängenden Herausforderungen der Softwareentwicklung mit Schwerpunkt auf Sicherheit und Zuverlässigkeit sowie Qualität in den Endprodukten diskutiert. Hochkarätige eingeladene Hauptredner waren Prof. Stefan Jähnichen, Past President der GI, der über „Modelica - Modeling and Simulation of Technical Systems“ referierte, Dr. Matthias Klauda, Department Head, Automotive Systems Integration (C/AI), Robert Bosch GmbH, der einen Einblick in die „Automotive Safety and Security from a Supplier’s Perspective“ gab, sowie Dr. Stefan Schmerler, Spokesperson of the International AUTOSAR Consortium und Director of Advanced Engineering of E/E Architecture and Standards der Daimler AG, Böblingen, der über „Evolution of Functional Safety & Security in AUTOSAR” sprach. Direkt vor der Konferenz fanden am 13.11. halbtägige Tutorial von Dr. D. L. Buttle, ETAS GmbH, Stuttgart über „AUTOSAR and Functional Safety“, von Prof. Dr. Lambert Grosskopf, Universität Bremen, über „Rechte an Geo- und Telemetriedaten“, von Dipl.-Ing.(BA) Christian Wenzel-Benner, ITK Engineering AG, Herxheim über „AUTOSAR & Security“ sowie von Jost Bracher und Frank Böhland, Robert Bosch GmbH, über „Safer C / Coding Pitfalls“ statt. Parallel zur Konferenz zeigte eine Ausstellung einschlägige Werkzeuge für den Automotive Bereich.

Softwarebasierte Funktionen stellen den wesentlichen Mehrwertfaktor in nahezu allen Teilen der Industrie, und insbesondere im automotiven Sektor, dar. Mit vertretbaren Kosten können durch Software erhebliche technische Fortschritte erzielt werden. Wesentlich ist dabei aber, die Zuverlässigkeit gerade in eingebetteten Systemen trotz der wachsenden Komplexität auf anspruchsvollem Niveau zu erhalten. In frühen Phasen eingeführte Softwarefehler dürfen in der Wartung nicht zur Kostenexplosion und zu unvertretbaren Verzögerungen führen. Hier müssen verbesserte Vorgehensweisen, Methoden und Werkzeuge zur Systematisierung des Entwicklungsprozesses und zur Gewährleistung der nötigen Sicherheit und Zuverlässigkeit eingesetzt werden. Hinzu kommt die hochgradige Vernetzung im und außerhalb des Fahrzeugs. Kritische Ereignisse in letzter Zeit führen drastisch vor Augen, dass ohne die Berücksichtigung entsprechender Sicherheitsanforderungen erhebliche Risiken in der zukünftigen Entwicklung drohen. Tool Chains, statische Verifikation, Partitionierung, Memory Management, Multi-Core und modellbasierte Vorgehensweisen sind einige der betrachteten Themen, die in diesem Zusammenhang eine wesentliche Rolle spielen und allen Beteiligten, ob Forschung, Entwicklung oder Akademia, am Herzen liegen.

Diese Interessenlage spiegelte sich auch im Spektrum und den Themen der Einreichungen zu dieser Tagung wieder. An dieser Stelle sei dem Programmkomitee gedankt, das trotz der hohen Zahl von Einreichungen zu den nötigen vielen Reviews bereit war, um eine solide Abdeckung bei der Begutachtung aller Beiträge zu gewährleisten. Aus der Menge der eingereichten Papiere wurden sorgfältig die besten

und einschlägigsten Beiträge zur Publikation und zum Vortrag ausgewählt. Leider konnten nur weniger als die Hälfte der Einreichungen im Zeitrahmen der Konferenz platziert werden. In sieben Sessions wurden die angenommenen Beiträge dem Publikum präsentiert. Die Schlusssession bot die Gelegenheit, Awards für das Best Paper und die Best Presentation zu vergeben.

Die Tagung ist Teil einer Serie, die seit 2004 im zweijährigen Turnus veranstaltet wird. Die Tagungsbände der Serie sind Zeitzeugen eines hochinteressanten Wandels der Interessen rund um software-gestützte Funktionen im Automobil. Unverkennbar ist der Trend über die Jahre hinweg, den schwierigen Weg zu hoher Zuverlässigkeit und Sicherheit auf systematischeren und moderneren Wegen als bisher üblich zu meistern und sich dabei stärker mit einschlägigen Standards auseinanderzusetzen. Mit der diesjährigen Konferenz hoffen wir, zur weiteren Verbreitung des dafür notwendigen Wissens beigetragen zu haben.

Die Abendveranstaltung kombinierte das Konferenzdinner mit einer Rheinrundfahrt, die auf dem Fahrgastschiff MS Karlsruhe im Rheinhafen Karlsruhe ihren Anfang und, zu später Stunde, auch ihr Ende fand. Die abendliche Dinnerrede hielt Prof. Schreckenberg über “Braucht das Auto den Menschen“ – eine wohl treffende Frage für die Zukunft.

Die fachlichen Träger der Automotive Tagung sind die Fachgruppen Ada, ENCRESS und EZQN der Gesellschaft für Informatik in den Fachbereichen "Sicherheit - Schutz und Zuverlässigkeit" und "Softwaretechnik" sowie der Fachausschuss "Embedded Software" der VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik und der Förderverein Ada Deutschland e.V. Die Tagungsleitung hatten Hubert B. Keller, Karlsruher Institut für Technologie; und Silke Spitzer, Robert Bosch GmbH, inne. Die wissenschaftliche Leitung erfolgte durch Hubert B. Keller, Karlsruher Institut für Technologie; und Erhard Plödereder, Universität Stuttgart. Der Dank geht an das Organisationsteam bestehend aus Peter Dencker, ETAS GmbH (Ausstellung), Reiner Kriesten, Hochschule Karlsruhe (Tutorial), Sabine Scheer, KIT (Tagungssekretariat), Herbert Klenk, Cassidian (Finanzen) und Silke Spitzer, Robert Bosch GmbH (Organisation vor Ort).

Die organisierenden Gruppierungen trafen sich am Montagabend zu ihren Versammlungen, anschließend fand ein Come Together statt.

Der Dank der Tagungsleitung geht an Silke Spitzer und die Robert Bosch GmbH für die Unterstützung bei der Durchführung, an AdaCore für die finanzielle Unterstützung, an Frau Cornelia Winter, Gesellschaft für Informatik, für die zügige Weiterleitung der Proceedingsunterlagen, sowie an H.C. Mayr für die Aufnahme dieses Tagungsbands in die LNI Reihe. Dank auch an den Förderverein Ada Deutschland e.V. für die Übernahme der finanziellen Verantwortung für die Gesamtveranstaltung.

Karlsruhe, den 15. November 2012

Hubert B. Keller Erhard Plödereder

Tagungsleitung
Hubert B. Keller, Karlsruher Institut für Technologie (KIT)
Silke Spitzer, Robert Bosch GmbH

Wissenschaftliche Leitung
H. B. Keller, Karlsruher Institut für Technologie (KIT)
E. Plödereder, Universität Stuttgart

Organisation
Peter Dencker, ETAS GmbH
Herbert Klenk, Cassidian
Reiner Kriesten, Hochschule Karlsruhe
Sabine Scheer, Karlsruher Institut für Technologie (KIT)
Silke Spitzer, Robert Bosch GmbH

Programmkomitee
Hubert B. Keller (Vorsitz), Karlsruher Institut für Technologie (KIT)
Erhard Plödereder (Vorsitz), Universität Stuttgart
Gerhard Beck, Rohde & Schwarz SIT GmbH
Manfred Broy, TU München
Stefan Bunzel, Continental
Peter Dencker, ETAS GmbH
Dirk Dickmanns, EADS
Claudia Eckert, TU München
Martin Emele, Robert Bosch GmbH
Simon Fürst, BMW AG
Peter Göhner, Universität Stuttgart
Ursula Goltz, TU Braunschweig
Klaus Grimm, Daimler AG
Erwin Großpietsch, EUROMICRO
Albert Held, Daimler AG
Stefan Jähnichen, TU Berlin
Herbert Klenk, Cassidian
Reiner Kriesten, Hochschule Karlsruhe
Thomas Kropf, Universität Tübingen
Ulrich Lefarth, Thales Transportation Systems
Jürgen Mottok, Hochschule Regensburg
Harald Neumann, Robert Bosch GmbH
Francesca Saglietti, Universität Erlangen-Nürnberg
Jörn Schneider, Fachhochschule Trier
Elmar Schoch, Volkswagen AG
Claus Stellwag, Elektrobit Automotive GmbH
Hans-Jörg Wolff, SB LiMotive GmbH

Veranstalter der Automotive 2012
Gesellschaft für Informatik mit den Fachgruppen Ada, ENCRESS, EZQN

VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik mit dem FA 5.11 Embedded Software, Förderverein Ada Deutschland e.V.

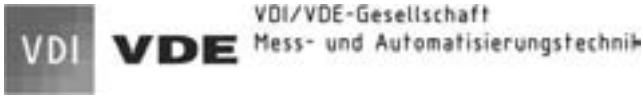
Sponsoren

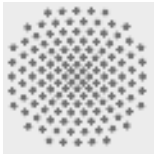
BOSCH
Invented for life

FG ENCRESS

FG "Sicherheit - Schutz und Zuverlässigkeit"

FG Ada





Universität Stuttgart

Inhaltsverzeichnis

Hauptvorträge

Stefan Jähnichen
Modelica - Modeling and simulation of technical systems……. 11

Matthias Klauda
Automotive Safety and Security from a Supplier's Perspective…….. 13

Stefan Schmerler
Evolution of Functional Safety & Security in AUTOSAR…… 23

Vorträge

Oscar Slotosch, Martin Wildmoser, Jan Philipps, Reinhard Jeschull, Rafael Zalman
ISO 26262 - Tool chain analysis reduces tool qualification costs…….…….. 27

Daniel Kästner, Christian Ferdinand
Static Verification of Non-Functional Software Requirements in the ISO-26262... 39

Martin Wittiger, Steffen Keul
Extraktion von Interthread-Kommunikation in eingebetteten Systemen……..…… 55

Dirk Nowotka, Johannes Traub
MEMICS - Memory Interval Constraint Solving of (concurrent) Machine Code.. 69

David Haworth, Tobias Jordan, Alexander Mattausch, Alexander Much
Freedom from Interference for AUTOSAR-based ECUs: a partitioned AUTOSAR stack……. 85

Jörn Schneider
Why current Memory Management Units are not suited for Automotive ECUs…. 99

Klaus Becker, Christian Buckl, Alexander Camek, Rainer Falk, Ludger Fiege, Jürgen Geßner, Stephan Sommer
SW-basierte Integration von neuen Fahrzeugfunktionen in zentralisierten Controllern……. 115

Cyrille Comar, Johannes Kanig, Yannick Moy
Integration von Formaler Verifikation und Test…… 133

Christian Berger, Sebastian Siegl
Constructive Requirements Modeling – More Reliable Implementations in a Shorter Time…… 149

Jürgen Braun, Jürgen Mottok, Christian Miedl, Dirk Geyer, Mark Minas
Increasing the reliability of single and multi core systems with software rejuvenation and coded processing…… 163

Rasmus Adler, Sören Kemmann, Markus Schurius, Christian Allmann
Modellbasierte Sicherheitsanalysen im BMBF-Förderprojekt e performance…. 179

Marko Wolf, Michael Scheibel
A Systematic Approach to a Qualified Security Risk Analysis for Vehicular IT Systems…….. 195

Felix Jakob, Wolfgang Kremer, Andreas Schulze, Jürgen Großmann, Nadja Menz, Martin Schneider, Allain-Georges Vouffo Feudjio
Risk-based testing of Bluetooth functionality in an automotive environment…… 211

Stefan Kriso
ISO 26262 - Quo vadis?..... 229

Hauptvorträge

Modelica - Modeling and Simulation of Technical Systems

Stefan Jähnichen, Peter Pepper
Technische Universität Berlin

stefan.jaehnichen@tu-berlin.de
peter.pepper@tu-berlin.de

Abstract: The talk introduces the simulation language Modelica and its concepts for the modeling and simulation of continuous system behavior. The language is competitive to Matlab/Simulink and in addition, supports an object-oriented style of modeling which leads to a high degree of component reuse. System behavior is expressed in an equation oriented style and algebraic and differential equations are directly encoded in the program text based on its modular structure. The talk gives an overview on the language concepts and on the mathematics as the interface to the solvers needed for experimentation, and introduces the advanced concept of structure dynamics . As a nice and novel feature, simulation results can be presented through a sophisticated graphical backend attached to the simulation system. Emphasis is also put on the concept of acausalty as a comfortable way to describe problems not only for the simulation of technical systems.

Automotive Safety und Security aus Sicht eines Zulieferers

Matthias Klauda, Stefan Kriso, Reinhold Hamann, Michael Schaffert

Zentralabteilung Automotive Systems Integration (C/AI)
Center of Competence „Functional Safety“
Center of Competence „E/E-Architecture“
Robert Bosch GmbH
Robert-Bosch-Allee 1
74232 Abstatt
{matthias.klauda, stefan.kriso, reinhold.hamann2, michael.schaffert}@de.bosch.com

Abstract: Sichere (im Sinne von „safe“) Systeme im Automobil zu entwickeln und zu produzieren, ist seit Jahrzehnten etablierter Stand der Technik. In jüngster Vergangenheit wurde zur Umsetzung eines konsolidierten Ansatzes zur funktionalen Sicherheit in Straßenfahrzeugen eine neue Norm – die ISO 26262 – erarbeitet. Diese Norm trägt insbesondere den wachsenden Herausforderungen immer komplexerer Systeme und neuer Technologien Rechnung. Allerdings sind Teile der ISO26262 bewusst offen oder visionär formuliert, so dass für eine sinnvolle Umsetzung in vielen Fällen eine einheitliche Interpretation innerhalb der Automobilindustrie unabdingbar ist.

Mit steigender Vernetzung der Fahrzeuge sowohl fahrzeugintern als auch mit der Umgebung (Car-2-X-Kommunikation) bietet das Automobil immer mehr Angriffspunkte für externe Attacken, so dass das Thema Automotive Security wachsende Bedeutung gewinnt.

Einerseits gibt es sowohl bei den Entwicklungsprozessen als auch in der technischen Implementierung Synergien, andererseits aber auch konkurrierende Aspekte zwischen Safety und Security. Dies macht eine enge Zusammenarbeit zwischen diesen beiden Domänen notwendig, um mögliche Synergien zu heben sowie die konkurrierenden Aspekte beherrschen zu können. Zum anderen erscheint es sinnvoll, ein gemeinsames Branchenverständnis im Bereich der Security zu schaffen und hierdurch im Sinne der Sicherheit des Kunden erprobte Methoden und Konzepte branchenweit einheitlich zu etablieren. .

1 Motivation

Safety und Security – im Deutschen beides mit Sicherheit übersetzt – stellen aktuell wichtige Treiber in der Entwicklung elektrischer/elektronischer Systeme im Automobil dar. „Safety“ adressiert hierbei den Schutz des Benutzers (Fahrer, Mitfahrer, Passanten,

etc.) vor dem Fehlverhalten des Systems, d.h. von der Benutzung des Systems darf keine größere Gefährdung ausgehen als es nach dem Stand von Wissenschaft und Technik vermeidbar ist [BGH09]. Während „Safety“ also die Sicht auf das System von innen nach außen darstellt, betrachtet die „Security“ den umgekehrten Fall, die Sicht auf das System von außen nach innen: Hier geht es darum, das System vor unbefugter Benutzung und Manipulation (Hacker, Produzent von Billigersatzteilen, etc.) zu schützen. Ein einfaches Beispiel für ein automobiles System, das sowohl Safety als auch Security adressiert ist ein einfaches mechanisches Türschloss: Im Normalbetrieb soll es das Fahrzeug, seine Werte und seine Insassen durch unberechtigten Zugriff von außen schützen (Security), im Falle eines Unfalls soll es jedoch ein leichtes Öffnen der Tür sicherstellen, um Rettungsmaßnahmen zu ermöglichen (Safety). Die Implementierung dieser beiden Aspekte kann sowohl mechanisch als auch elektronisch erfolgen. In diesem Beitrag wollen wir uns jedoch beschränken auf die elektrischen/elektronischen Systeme, d.h. im Bereich der Safety auf die „Funktionale Sicherheit“ (d.h. auf die von Fehlfunktionen elektrisch/elektronischer Systeme ausgehenden Gefährdungen), im Bereich der Security auf die „Datensicherheit“ (d.h. z.B. nicht auf die Panzerung von Fahrzeugen).

Das Thema Safety ist für die Automobilindustrie nichts Neues; sichere Systeme (im Sinne von „safe“) zu entwickeln und zu produzieren ist seit vielen Jahrzehnten Stand der Dinge. In weltweiten Unfallstatistiken spielen technische Mängel von elektrischen/elektronischen System als Unfallursache eine untergeordnete Rolle (< 1%, siehe z.B. [Li10]).

Dagegen ist der Bereich Security eine recht junge Disziplin. In der Vergangenheit war das Thema zwar schon in einzelnen Domänen adressiert (Tuningschutz für Steuergeräte), jedoch erst durch die zunehmende Realisierung von Fahrzeugfunktionen über Software und die immer mehr aufkommende Vernetzung von Fahrzeugen mit ihrer Umwelt (Internet of Things) gewinnt der Zugriffsschutz eine wachsende Bedeutung.

Eine besondere Herausforderung entsteht darüber hinaus durch die Verbindung der beiden Themen Safety und Security, d.h. einerseits Synergien zwischen beiden zu nutzen, andererseits aber auch sich gegenseitig konterkarierende Maßnahmen zu verhindern.

2 Automotive Safety

Bei der Entwicklung von elektrischen/elektronischen Systemen steht die funktionale Sicherheit schon seit langem im Fokus der Entwicklung. Die Automobilindustrie hat in der Vergangenheit gelernt, wie funktional sichere elektrische/elektronische Systeme zu entwickeln und zu produzieren sind. Wachsende Bedeutung erlangt das Thema in jüngster Vergangenheit durch verschiedene Treiber:

- Die Hochintegration elektronischer Bauelemente (kleinere Strukturgrößen) führt zu einer wachsenden Anfälligkeit gegenüber äußeren Einflüssen.

- Durch steigende Komplexität der Systeme an sich besteht die Gefahr ungewollter bzw. unvorhersehbarer Wechselwirkungen zwischen den Teilkomponenten, z.B. bei neuartigen Funktionen, die rein in Software dargestellt und möglicherweise auf mehrere Steuergeräte verteilt sind.

- Neue Technologien wie zum Beispiel die Elektrifizierung des Antriebstrangs führen zu neuartigen Safety-Fragestellungen, mit denen die Automobilbranche bisher nicht in diesem Maße konfrontiert war.

- Die Tatsache, dass es keine absolut sicheren Systeme gibt, setzt jeden Hersteller der Gefahr aus, dass seine spezifische Lösung im Ernstfall als nicht ausreichend eingestuft wird. Dies hat immer mehr Hersteller einsehen lassen, dass diese Produkthaftungsrisiken nur durch einen gemeinsam definierten Sicherheitsstand zu minimieren sind.

Um diesem Rechnung zu tragen, wurde eine neue Norm zur funktionalen Sicherheit speziell von elektronischen Systemen in Straßenfahrzeugen erarbeitet und im November 2011 veröffentlicht - die ISO 26262 [ISO11]. Diese wurde von den Automobilherstellern und -zulieferern gemeinsam erarbeitet und beschreibt als branchenspezifische Ableitung der IEC 61508 auf einer relativ hohen Abstraktionsebene, wie sicherheitsrelevante Systeme im Automobil entwickelt werden (sollen).

Da aus formaljuristischer Sicht die ISO 26262 zum Stand der Technik beiträgt, ist es zur Reduktion unberechenbarer Produkthaftungsrisiken dringend angeraten, die ISO 26262 umzusetzen, auch wenn es hierzu keine direkte gesetzliche Verpflichtung gibt. Die auf Grund ihres Abstraktionsgrades notwendige Interpretation führt hier aber dazu, dass unterschiedliche Norminterpretationen aufeinandertreffen, die der Produktsicherheit teilweise sogar eher ab- als zuträglich sein können.

Als Beispiel sei an dieser Stelle die Unabhängigkeit bei einem Functional Safety Assessment genannt: Einerseits wird für ein System mit höchstem Automotive Safety Integrity Level (ASIL D) gefordert, dass ein solches Assessment von einer Person durchgeführt wird, die organisatorisch unabhängig ist vom verantwortlichen Projekt. Andererseits soll das Assessment aber die erreichte funktionale Sicherheit des Produkts bewerten, d.h. es soll die Wirksamkeit und die Angemessenheit der implementierten Sicherheitsmaßnahmen beurteilen. Hierzu ist detailliertes Produktwissen notwendig. Dies steht üblicherweise im Widerspruch zur geforderten Unabhängigkeit, da die Fachexperten mit detailliertem Produktwissen eben nicht aus einer anderen, sondern oftmals aus derselben Organisationseinheit kommen. Wird nun die Unabhängigkeitsanforderung der ISO 26262 in dem Sinne überinterpretiert, als dass sie ausschließlich durch externe Assessoren dargestellt werden kann, besteht die Gefahr, dass das Assessment auf Grund mangelnden Produktwissens wertlos ist.

Ein unterschiedliches Branchenverständnis führt hier nicht nur zu Inkompatibilitäten an Zuliefererschnittstellen, sondern unter Umständen auch zu Produkthaftungsproblemen bei Abweichungen vom Branchenverständnis „nach unten“ oder zu unnötigem Overdesign ohne Sicherheitsgewinn bei Abweichungen „nach oben“ [KHK11, KH11].

Insbesondere Automobilzulieferer stehen vor der Herausforderung, unterschiedliche Anforderungen verschiedener OEM umsetzen zu müssen. Beispielsweise stellt sich bei einer Komponenten-Plattformentwicklung die Frage, nach welchem Sicherheitsintegritätslevel (ASIL) diese sinnvoll entwickelt werden kann, um einerseits möglichst viele Kundenforderungen bedienen zu können, um sich aber andererseits auf ein sinnvolles Maß an Aufwand zu beschränken und kein Overdesign ohne Sicherheitsgewinn zu betreiben. Ein gemeinsames Branchenverständnis – durch Erarbeitung eines gemeinsamen Standards, aber auch durch Bilden einer gemeinsamen Interpretation – stellt sicher, dass ein flächendeckend gleiches Niveau an Sicherheit (Safety) gewährleistet wird.

Um dieses Branchenverständnisses weiter zu harmonisieren, hat die Automobilbranche bereits begonnen, an der 2nd Edition der ISO 26262 zu arbeiten, deren Veröffentlichung gegen 2016/2017 zu erwarten ist. Es ist damit zu rechnen, dass hier das Thema „Automotive Security“ zumindest erwähnt werden wird, da Security eine der wichtigen Voraussetzungen für Safety darstellt.

3 Automotive Security

Die stark anwachsende Komplexität elektrischer/elektronischer Systeme – sowohl innerhalb eines Fahrzeugs an sich als auch außerhalb durch dessen Vernetzung mit seiner Umwelt – schafft immer mehr mögliche Angriffspunkte für Angriffe von außen. Insbesondere bietet die drahtlose Vernetzung (GSM, WLAN, Bluetooth) neue Möglichkeiten des (unautorisierten) Zugriffs auf das Fahrzeug, so dass es nicht mehr in dem Maße wie in der Vergangenheit als ein in sich geschlossenes System betrachtet werden darf. Angriffe auf das Fahrzeug können aus verschiedenen Motivationen heraus erfolgen:

- Angreifer, die versuchen, unautorisierten Zugriff auf das Fahrzeug oder seine Komponenten zu erlangen und Korruption / Deaktivierung einzelner Fahrzeugfunktionen androhen (Angriff auf „Availability of Service“)

- „Tuning“ des Fahrzeugs durch Verändern von Eigenschaften/Funktionen (z.B. Chip-Tuning, Fälschen des Tachometerstandes oder Deaktivieren von störenden Warnmeldungen)(Angriff auf „Functional Integrity“),

- Angriff auf die Fahrzeug-Infrastruktur, um darüber an personenbezogene Daten zu gelangen, die z.B. auf einem mit dem Fahrzeug vernetzen Smartphone gespeichert sind (Angriff auf „Personal Integrity“),

Bisher stand die Security einzelner Steuergeräte und ihrer Daten im Vordergrund. Hierfür wurden z.B. in Form des Bosch Hardware Security Moduls (HSM) Lösungen gezeigt und implementiert. Dieses bietet Unterstützung bei der Absicherung folgender Funktionalitäten:

- Secure Flashing: Absicherung, dass nur korrekte Software auf ein Steuergerät heruntergeladen werden kann.

- Secure Boot: Absicherung, dass nur korrekte Software zur Ausführung gebracht werden kann.

- Run-Time Tuning Detection: Absicherung, dass die Ausführung nicht korrekter Software erkannt wird.

- Secure Debug: Absicherung, dass kein Angriff über das Debug-Interface des Mikrocontrollers erfolgen kann.

Im Gegensatz zu üblicherweise separaten „Trusted Platform Modules“ (TPM) ist dieses On-Chip Hardware Security Module (HSM) integriert auf dem Mikrocontroller.

Die zunehmende Vernetzung der Steuergeräte untereinander sowie die Vernetzung mit der Fahrzeugumgebung macht jedoch ebenfalls die Absicherung der Kommunikation *zwischen* den Steuergeräten notwendig: Beispielsweise können Steuergeräte heute in der Regel nicht feststellen, ob die im richtigen Format erhaltene BUS-Nachricht auch vom richtigen Steuergerät kommt oder ein Angreifer das sendende Steuergerät simuliert.

Angriffe mit dem Ziel, auf breiter Front bewusst sicherheits-(safety-)kritisches Verhalten zu provozieren, erscheinen aus heutiger Sicht zwar unwahrscheinlich [We12], können für die Zukunft aber nicht ausgeschlossen werden. Daneben können natürlich auch alle anderen Security-Angriffe auf das Fahrzeug safety-kritisches Verhalten auslösen, auch wenn dies vom Angreifer primär nicht gewollt ist. Vor diesem Hintergrund stellt Security eine wesentliche Voraussetzung für Safety dar; ohne geeignete Security-Maßnahmen lassen sich viele Safety-Funktionen nur schwer umfassend absichern. Kann eine zu einfache Manipulation des Fahrzeugs gar als „zu erwartender Missbrauch“ betrachtet werden, ist es fast schon als Stand der Technik anzusehen, Safety durch Security-Maßnahmen sicherzustellen.

Neben der reinen Absicherung der „Safety Integrity“ ist ein weiterer Treiber die Zuverlässigkeit bzw. Verfügbarkeit des Systems an sich. So lassen sich durch Security-Maßnahmen auch nicht safety-kritische Manipulationen am Fahrzeug erkennen und verhindern – wie zum Beispiel die Verwendung nicht zugelassener Ersatzteile mit deutlich verminderter Performance oder Lebensdauer (Produktpiraterie).

Deutlich wird, dass es - wie auch für Safety - auch für Security kein absolutes Maß für Sicherheit gibt. Jede individuelle Lösung kann dem Vorwurf ausgesetzt sein, nicht vollständig sicher zu sein. Ohne eine branchenweite Abstimmung und damit einem gemeinsamen Ringen um einen allgemeingültigen Stand der Wissenschaft und Technik setzt sich jeder Marktteilnehmer einem hohen Risiko aus, dass seine Lösung nach Jahren im Feld als unzureichend bewertet wird.

4 Gemeinsame Betrachtung von Safety und Security

Um Synergien zwischen Safety und Security nutzen zu können, muss man die Frage stellen, wo Gemeinsamkeiten liegen. Diese findet man zum einen im Entwicklungsprozess, zum anderen aber auch in den technischen Lösungsansätzen.

4.1 Entwicklungsprozess

Während es in der ISO 26262 für Safety einen beschriebenen Entwicklungsprozess gibt, findet man diesen für Security in dieser Form bislang nicht. Genauere Betrachtung zeigt jedoch auf einer entsprechenden Abstraktionsebene viele Gemeinsamkeiten. Es erscheint durchaus sinnvoll, die beiden Entwicklungsprozesse miteinander zu verweben [Eh10, Bu12]. Abbildung 1 schlägt einen Entwicklungsprozess vor, der die Aspekte Safety und Security gemeinschaftlich adressiert:

Nach einer Festlegung des Betrachtungs- / Entwicklungsgegenstandes („Item Definition“) werden in einer Gefährdungsanalyse und Risikobewertung („Hazard Analysis and Risk Assessment“) die Gefährdungen aus Safety-Sicht ermittelt, die Sicherheitsrelevanz (ASIL = Automotive Safety Integrity Level) festgelegt, die umzusetzenden Sicherheitsziele („Safety Goals“) abgeleitet und Maßnahmen („Safety Measures“) zur Umsetzung der Sicherheitsziele definiert. Analog hierzu erfolgt auf der Security-Seite in einer Bedrohungsanalyse („Security Rrisks Analysis“) die Ermittlung möglicher Bedrohungsszenarien, woraus ebenfalls Sicherheitsziele („Security Objectives“) und dementsprechende Maßnahmen („Security Measures“) abgeleitet werden.

Nach Design und Implementierung des Systems sowie dessen Verifikation und Validierung erfolgt einerseits ein „Functional Safety Assessment“, das die erreichte funktionale Sicherheit bewertet, andererseits auf der Security-Seite ggf. eine Begutachtung der erreichten Bedrohungssicherheit.

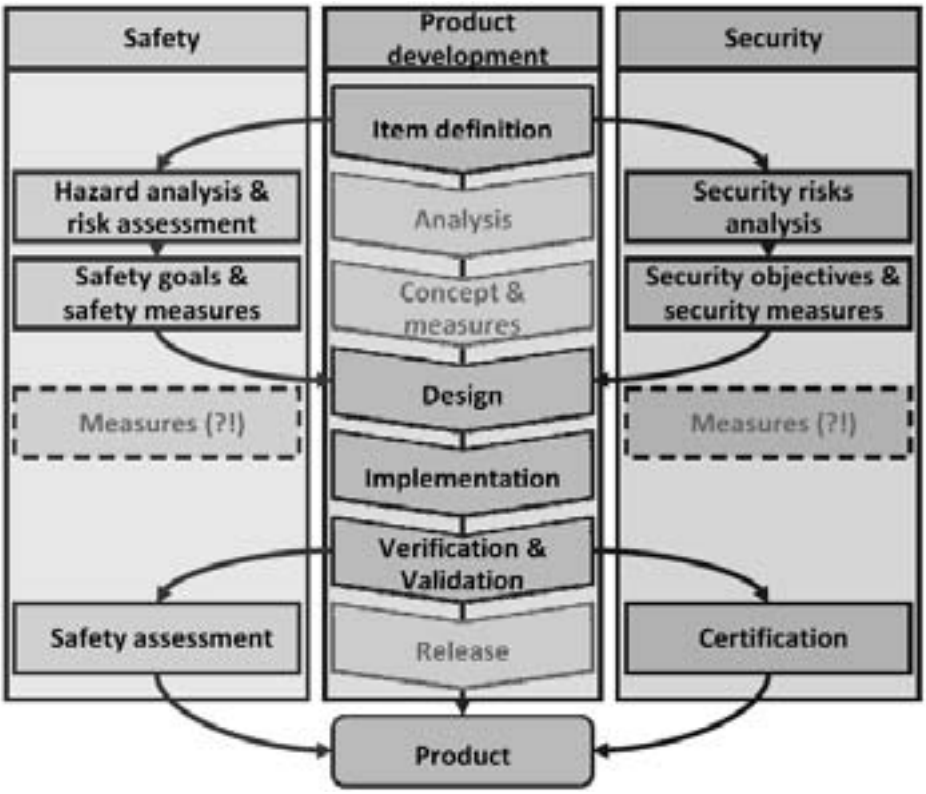


Abbildung 1: Vorschlag für die Berücksichtigung von Safety und Security in einem gemeinsamen Entwicklungsprozess [Eh10]

Auf Grund der Ähnlichkeit beider Vorgehensweisen, der sich daraus ergebenden Synergien sowie des fortgeschrittenen Status des Safety Lifecycles in der bereits veröffentlichten ISO 26262 wurde in [Bu12] vorgeschlagen, die Methodik ISO 26262 um den Aspekt der Security zu erweitern. Inwieweit diese Idee jedoch dem Thema Security gerecht wird, oder ob andere Ansätze eher zielführend sind, muss detailliert evaluiert werden. Eine Community-weite Standardisierung des Themas Security, wie es bei der Safety mit der ISO 26262 gelungen ist, erscheint jedenfalls notwendig und sinnvoll.

4.2 Technische Abhängigkeiten

Nicht nur auf der Ebene der Entwicklungsprozesse und der Methodiken gibt es Abhängigkeiten, sondern auch auf technischer Ebene. Als Beispiele, in denen Security-Lösungen die Safety des Produkts erhöhen können, seien genannt [We12]:

- Sicheres Software-Update und Hochfahren des Systems, um Manipulationen am Steuergerät (Chip-Tuning) zu verhindern. Außerdem lässt sich dadurch das

Einschleusen von „Malware“ (z.B. Viren) verhindern, von welcher weitere Angriffe auf das System ausgehen könnten.

- Komponentenidentifikation durch Authentifizierung sichert ab, dass nur Original-Hardwarekomponenten verbaut werden und keine (möglicherweise gefälschten) Billigkomponenten mit verminderter Performance zum Einsatz kommen.

- Logische Trennung von Infotainment und Safety-relevanten Bereichen im Fahrzeug sichert ab, dass stattfindende Angriffe auf den nicht (oder gering) sicherheitsrelevanten Teil des Systems beschränkt bleiben.

- Absicherung der Kommunikationsbusse erlaubt die Erkennung manipulierter oder ausgeblendeter Botschaften auf Grund eines ungewollten Angriffs auf die Datenkommunikation an sich oder auf Grund einer Manipulation in einem Steuergerät.

Es ist daher notwendig, bei der Implementierung von Safety-Mechanismen auch die Security-Aspekte zu berücksichtigen und umgekehrt. Neben dem organisatorischen Aspekt, dass hierzu eine enge Vernetzung der Experten beider Domänen notwendig ist muss dies auch in der technischen Lösung Niederschlag finden. So mag zum Beispiel das Konzept der Virtualsierung (Hypervisor) geeignet sein, sowohl Safety-Funktionen von nicht-sicherheitsrelevanten Funktionen zu kapseln, gleichzeitig aber auch die Security des Safety-relevanten Teils sicherzustellen.

4.3 Konkurrierende Aspekte

Eine gemeinsame Betrachtung von Safety und Security ermöglicht auf der einen Seite die Nutzung von Synergien, ist aber auf der anderen Seite auch zwingend notwendig, um zu verhindern, dass durch konkurrierende Aspekte die Safety oder die Security des Systems kompromittiert werden.

Ein „Zuviel“ an Security kann beispielsweise Auswirkungen auf die Safety des Produkts haben. So könnte eine Fehlfunktion in der Security-Funktion das Deaktivieren einer Safety-Funktion oder –Komponente zur Folge haben. Wie zum Beispiel in [St12] ausgeführt ist es hier wichtig darauf zu achten, dass als Handlungsleitlinie die Security-Maßnahmen wichtige Safety-Funktionen nicht beeinträchtigen dürfen und die Verfügbarkeit der Safety-Funktion hier gegenüber der Security-Funktion Vorrang haben muss.

5 Zusammenfassung

Bei der Betrachtung von Automotive Safety und Automotive Security darf man sich nicht darauf beschränken, beide Themen unabhängig voneinander zu betrachten.

Zum einen ist es notwendig, Safety und Security gemeinschaftlich zu bearbeiten, um vorhandene Synergien nutzen zu können. Im Bereich der Entwicklungsprozesse heißt das, Vorgehensweisen und Methoden zu finden und in der Organisation zu verankern, die ein gemeinsames Bearbeiten der beiden Themenfelder erlaubt und fördert; dies stellt in erster Linie eine organisatorische Herausforderung dar, da es heute noch nicht sichergestellt ist, dass Safety- und Security-Experten um ihre gegenseitigen Arbeitsfelder wissen und eine Zusammenarbeit systematisch erfolgt. Diese ist aber notwendig, um auch die Synergie auf technischer Ebene in der Implementierung des Systems heben zu können. Gleichzeitig ist dies aber auch nicht nur notwendig, um Synergien zu heben, sondern auch um Security-Maßnahmen, die zur Umsetzung von Safety-Anforderungen dienen, effizient implementieren und potenzielle Konflikte zwischen beiden Domänen verhindern zu können.

Analog zum Safety-Umfeld ist auch bei der Security eine Standardisierung und die Schaffung eines gemeinsamen Branchenverständnisses notwendig, da es ansonsten durch Reibungsverluste zwischen den beteiligten Stakeholdern eines weltweit verteilten OEM- / Zulieferernetzwerks dazu führen kann, dass Security-Lücken entstehen, in die potenzielle Angreifer eindringen und beispielsweise Safety-Probleme provozieren können. Wie beim Thema Safety besteht diese Notwendigkeit insbesondere dann, wenn die Systemsicherheit (Security) von mehreren Beteiligten - d.h. z.B. von Subsystemen verschiedener Zulieferer - abhängt und eine gemeinsame Abstimmung auf Basis eines einheitlichen Verständnisses notwendig wird.

Literaturverzeichnis

[BGH09] BGH Urteil v. 16.6.2009 VI ZR 107/08

[Bu12] Burton, S. et al.: Automotive Functional Safety = Safety + Security. In: 1st International Conference on Security of Internet of Things (SecurIT 2012), Kerala, Indien, 17.-19.08.2012

[Eh10] Eherer, S. et al.: Synergetic Safety and Security Engineering – Improving Efficiency and Dependability. In: 8th escar Embedded Security in Cars conference, Bremen, 16.-17.11.2010

[ISO11] International Organization for Standardization: ISO 26262:2011(E). Genf, 2011.

[KHK11] Klauda M.; Hamann R.; Kriso S.: ISO 26262 – Was kommt da auf uns zu? In: VDI-Berichte 2132, 2011, S. 285-297.

[KH11] Kriso S.; Hamann R.: Die ISO 26262 ist veröffentlicht - Konsequenzen für OEMs und Zulieferer. In: VDI-Berichte 2132, 2011, S. 299-308.

[Li10] Lich, T.: Unfallforschung, Möglichkeiten und Grenzen – Bezug zur ISO 26262. In: 2. Euroforum Jahrestagung zur ISO 26262, Stuttgart, 27.-28.10.2010.

[PW11] Pelzl, J.; Wolf, M.: Mehr Funktionssicherheit durch IT-Sicherheit! In: 3. Euroforum Jahrestagung zur ISO 26262, Stuttgart, 26.-27.09.2011.

[St12] Störtkuhl, T.: Securiy for Safety in der Industrieautomation – Bedrohungen, Schwachstellen, Risiken und Lösungsansätze. In: safe.tech 2012, München, 14.03.2012.

[We12] Weimerskirch A.: Functional Safety by Sound Data Security. In: CTI conference on ISO 26262, Detroit, 12.06.2012.

Evolution of Functional Safety & Security in AUTOSAR

Stefan Schmerler[1], Simon Fürst[2], Steffen Lupp[3], Demetrio Aiello[4], Frank Kirschke-Biller[5], Robert Rimkus[6], Alain Gilberg[7], Kenji Nishikawa[8], Andreas Titze[9]

[1] Daimler AG, Sindelfingen, [2] BMW AG, München, [3] Bosch GmbH, Heilbronn, [4] Continental AG, Regensburg, [5] Ford Motor Company, Köln, [6] General Motors, Detroit , [7] PSA, Paris, [8] Toyota Motor Company, Brüssel, [9] Volkswagen AG, Wolfsburg

stefan.schmerler@daimler.com

Abstract: AUTOSAR (AUTomotive Open System Architecture) is an open, international standard for the software architecture of automotive ECUs, which is commonly developed in an international consortium of several OEMs, tier1s, and software tool providers. Today, numerous series vehicles with AUTOSAR technology inside are on the road.

Within the AUTOSAR standard, several concepts and mechanisms to support safety & security were developed and included in the design of the AUTOSAR software architecture and in the corresponding functionality of the AUTOSAR basic software modules. Starting with its release 4.0 published in December 2009, AUTOSAR included enhancements with respect to safety-related applications in the automotive domain. The safety-related functionality of AUTOSAR and the functional safety standard ISO 26262 have been developed in parallel with mutual stimulation.

In relation to the described activities, an overview of the available safety & security functionality is shown and a brief description of the following concepts and specified mechanisms is provided:

- Built-in self-test mechanisms for detecting hardware faults (testing and monitoring)
- Run-time mechanisms for detecting software execution faults, e.g. program flow monitoring
- Run-time mechanisms for preventing interference between software elements, e.g. memory partitioning for software components and time partitioning for software applications
- Run-time mechanisms for protecting communication, e.g. end-to-end (E2E) communication protection
- Run-time mechanisms for error handling
- Crypto service manager
- Crypto abstraction library

Based on market needs, AUTOSAR plans to enhance the existing safety & security mechanisms and to support new methods and features in the future. An overview of the planned concepts and a brief description of the following extensions is provided:

- Integrated end to end protection
- Hardware test manager for tests at runtime
- Guide for the utilization of crypto services

In addition to the decribed concepts in the field of software architecture, AUTOSAR also plans to introduce several process and methodology improvements, which support the development processes with respect to safety & security aspects. The major ideas of the new conecpts are discussed and a brief description of the following improvements is provided:

- Tracability within the AUTOSAR specification documents
- Safety related extensions for the AUTOSAR methodology and templates
- Signal qualifier concept

14.-15. November 2012
5. Tagung
Karlsruhe
Automotive – Safety & Security 2012
Sicherheit und Zuverlässigkeit für automobile Informationstechnik
KIT
BOSCH
Invented for life
VDI
VDE

Vorträge

ISO 26262 - Tool Chain Analysis Reduces Tool Qualification Costs

Oscar Slotosch,
Martin Wildmoser,
Jan Philipps,
Reinhard Jeschull

Validas AG
München
<surname>@validas.de

Rafael Zalman

Infineon Technologies AG
Neubiberg
rafael.zalman@infineon.com

Abstract: Software tools in safety related projects are indispensable, but also introduce risks. A tool error may lead to the injection or non-detection of a fault in the product. For this reason the safety norm for road vehicles, ISO 26262, requires determination of a tool confidence level for each software tool. In this paper we present a model-based approach to represent a tool chain, its potential errors and the counter-measures for these. In this model tools are not only error sources, but can also act as error sinks for other tools by providing appropriate checks and restrictions. The tool work flow in a project can be rearranged or extended to make the integrated tool chain safer than its parts and to reduce tool qualification costs greatly. The tool chain model not only identifies the critical tools, but also exposes very specific qualification requirements for these. The paper illustrates and augments this approach with experiences and results from an application to a real industrial automotive tool chain consisting of 37 tools.

1 Introduction

The increasing visibility of development and safety standards such as the ISO 26262 is a sign of the growing maturity of the field of embedded software development. Growing maturity of an engineering field is quite naturally accompanied with a stronger emphasis on choice and suitability of tools. While some earlier standards demanded a kind of statement of "fitness for use", which was often provided through certification, the ISO 26262 takes a more holistic view and demands that tools are evaluated, and if necessary also qualified, in the context of their use in the development tool chain.
Thus, tool qualification in ISO 26262 [ISO26] is a two-step process. In the first step, tools are evaluated (see Fig 1) to determine a *tool confidence level* (TCL), based on the *tool impact* (TI) and on the *tool error detection* level (TD).

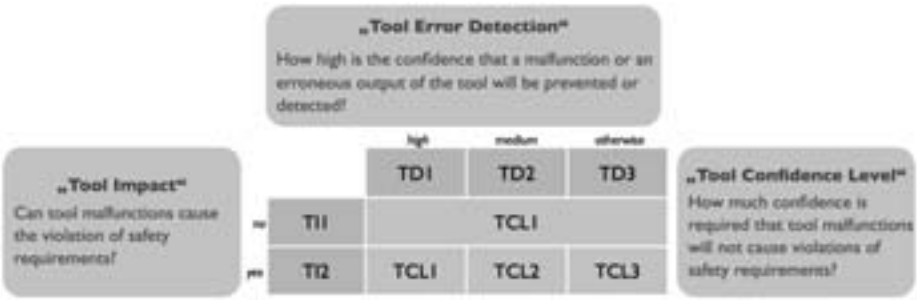


Fig 1. Tool evaluation according to ISO 26262:2011-8

The TI denotes the possibility of tool errors leading to a violation of a safety requirement and the TD expresses the likelihood of detecting or preventing tool errors.
In the second step, depending on the TCL and the safety integrity level of the development object, further qualification measures are demanded, such as for instance extensive and focussed tests (*tool validation*).
The safety norm ISO 26262 offers the possibility to avoid tool qualification by enriching the development process with additional checks and restrictions in order to detect or prevent all potential tool errors with high probability. With the additional checks and restrictions, the tool confidence level can sometimes be lowered to TCL1, where no tool qualification is needed.
In contrast to concrete tool errors, e.g. a bug in function X of tool Y, a *potential tool error* is a fictive and abstract error of a tool, e.g. loss of data in an output artifact, and denotes something that *can* go wrong in a tool of a certain type according to Murphy's law. Please also note that we use the term *tool error* to identify the occurrence of wrong tool behavior in a use case and not its cause, that is a *fault* or *defect* in the tool's code. In another paper [WPS12] we further distinguish between tool errors, which are internal to tool, and tool failures, which are external to tool and occur at the output side, according to Laprie's fault/error/failure model [L92], but in this paper we rather only use the term tool error for both in order to be more consistent with the ISO 26262 terminology.
Tool errors can be mitigated by counter measures in the form of checks or restrictions. A *check* is a counter measure that detects tool errors after they have occurred, e.g. by inspecting the tool's output artifacts. A *restriction* is a counter measure that prevents tool errors from occurring, e.g. by limiting the input space or used functions of a tool. Note that we usually cannot detect or prevent the cause of tool errors – i.e., tool faults –, as this would require measures on the tool's code or in its development process.
While there is a certain body of work on tool validation issues [SC04, SS07] and some work on specific tool use patterns to reduce the need of tool validation [Be10], there is so far little work that considers the holistic approach of establishing tool confidence according to ISO 26262. A notable exception is [Hi11], where the process of tool evaluation is explained in detail and some practical extensions are proposed.
Recently, an increasing emphasis on tool qualification can also be noticed in other safety norms besides the ISO 26262. The second edition of IEC 61508 [61508], the general safety norm for E/E/P systems, demands all software offline-support tools to be classified in levels T1-T3, but this is done for each tool in isolation and there is no way to

avoid qualification of T3 tools. Another example is DO-178C [DO178], the safety norm for the avionic domain, which also has a twofold strategy of first classifying tools in levels TQL1 to TQL5 and then qualifying them by designated activities described in DO-330 [DO330].
The classification in DO-178C is also based on the possibility of a tool to insert or fail to detect product errors, but is not dependent on the possibility of detecting or avoiding these tool errors elsewhere within the development process.

2 Tool Evaluation Process

Tool evaluation is about determining the TCL for all tools used in the development process. Our model-based tool evaluation process consists of the following steps:

(a) Model the tool chain with
- tools,
- use cases and
- artifacts.

(b) Determine tool impact for every use case

(c) Identify potential tool errors for every use case

(d) Identify and assign measures for error detection and –prevention

(e) Validate the model
- Automated Plausibility Checks
- Generation of tables for reviewing the tool chain model
- Rework the model according to the review results

(f) Generate a report including the computed tool confidence level for each tool

Step (a) results in a model of the tool chain to be analyzed. Before we discuss this model in detail let us look at an example. Assume a tool chain (see Fig 2) for testing product source code, e.g. software modules, with the aim of achieving 100% requirements coverage and 100% MC/DC structural coverage.
Our testing tool chain consists of three tools: a test automation tool (TAU), a compiler and an execution platform. The TAU is employed for the use case “RunTest”, which involves coverage instrumentation, test application generation and the call of other tools, that is compiler and platform, in order to execute a test case. The use case “RunTest” takes the product source files and a test case definition file (stimuli+references) as input and delivers as output an instrumented version of the product source files and the test application source code, which is the extra code needed for calling the functions to be tested with the inputs specified in the test case definition and to collect and compare the resulting values with the reference values from the test case definition.

The TAU calls the use cases “CompileApp” and “LinkApp” of the tool “Compiler” in order to produce the test application binary “TestApp ELF” and the use case “RunApp” of the tool “Platform” to execute the test application binary, e.g. by flashing it to an evaluation board, starting up the application and sending back the results via a communication software, e.g. terminal for COM messages.
The test output consists of verdicts of the form PASS or FAIL denoting the result of comparing the actual output values with the corresponding references and the coverage hits denoting the exercised parts of the product source code. The TAU receives these test results from the tool “Platform” and uses them to produce a summary test report.

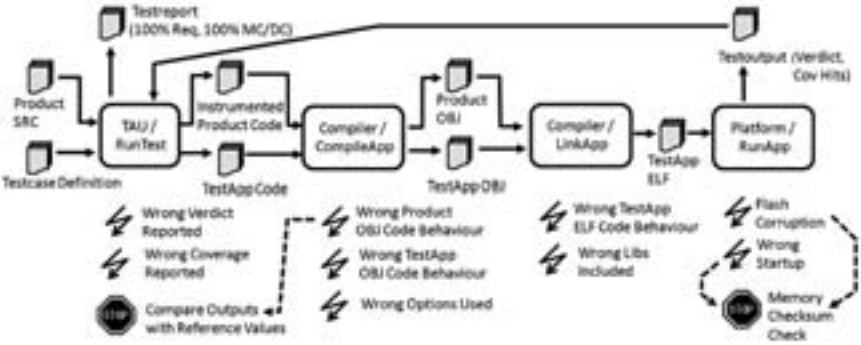


Fig 2. Example: Testing tool chain, contains some errors without counter measures

After the tool chain is modeled the next step (b) is to determine the tool impact for each use case. According to ISO 26262 a tool or use case has TI1 if there is no possibility that a potential tool error can lead to the injection or non-detection of a fault in the safety-related product, i.e. a violation of a safety requirement. Otherwise, if there is such a possibility or if the question cannot be answered the result is TI2.
In step (b) it is not only important to write down the decision (TI1, TI2), but also the reasons for this decisions. For example if a tool has no data flow to the safety related parts of the product and is not used for a verification step, then one can argue that such a tool is TI1.
For every use case having TI2 the next step (c) is to determine potential tool errors and then in step (d) to find possible counter measures for these if these exist.
In the testing tool chain we can identify several potential tool errors for the use cases, that might lead to the injection or non-detection of product faults. On the other hand there are also several possibilities to detect or avoid these potential errors within these use cases.
For example, the compiler may have a defect with leads to the injection of a fault into the product object file. However, this potential error “Wrong Product OBJ Code Behaviour” of use case “CompileApp” can very likely be detected by the check “Compare Outputs with Reference Values”, which is part of the use case “RunTest” of the tool TAU.

However, if a compiler defect hits the test application code, e.g. “Wrong TestApp OBJ Code Behaviour” in use case “CompileApp”, the test execution itself might get compromised and a product fault, either injected by a tool defect or manually inserted by the developer, might slip through. Also, if the TAU reports wrong coverage, i.e. potential error “Wrong Coverage Reported” of use case “RunTest”, it might not be noticed by the tester that parts of the product code have not been covered by tests and product faults residing in these unexercised parts might slip through.
After the tool chain model is adjusted with potential errors and counter measures the next step (d) is to validate this model, that is to check if it resembles correctly and completely the tool chain as it used in the development process and if the modeled errors and counter measures are feasible. In the end of the next section we will show how this step can be assisted with automatic plausibility checks and review tables generated from the model.

Once the model is validated the final step (e) is the computation of the TCL for each use case and tool and the generation of a report that contains the resulting TCLs and the way these are derived.

3 Meta Model for Tool Chains

To define and analyze tool chains we propose a simple domain specific tool chain model (see Fig 3), which consists of the following elements: Tool Chain, Tool, Use Case, Artifact, Error, Check, Restriction and Qualification.
The model shall represent two aspects:

- the tool chain structure consisting of tools, use cases and artifacts
- the tool confidence by representing the potential errors and assigned checks or restrictions

In the model a tool chain consists of tools, use cases and artifacts. Each tool has its distinct use cases, which describe the purpose for the tool within the project. Each use case has its own sets of artifacts, which are read, written or modified by the use case and can be assigned a TI (TI1, TI2).

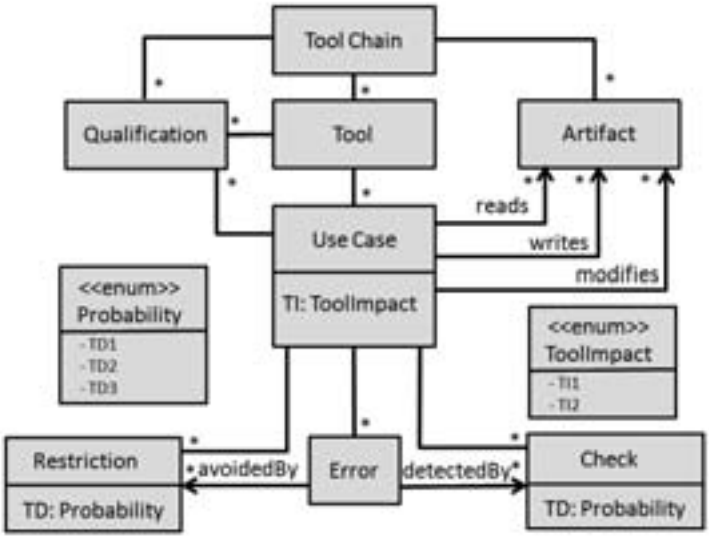


Fig 3. Domain Model for Tool Chains

To model the tool confidence situation each use case is associated with a set of potential errors, which may be assigned to checks or restrictions provided by the same or other use cases possibly from other tools. In our example tool chain (see Fig 2) the potential error "Flash Corruption" is assigned to the check "Memory Checksum Check" provided by the same use case "RunApp" from the tool "Platform", i.e. a simulator or a board with corresponding connection software.

Each check and restriction has its own effectiveness expressed by the TD level (TD1-TD3).

By using TI and TD for all identified potential errors, the TCL for a use case or a tool can be computed automatically according to the ISO 26262 table (see Fig 1) by propagating the maximum TD or TCL upwards.

Qualifications (including a selectable qualification method) can be added to tool chains, tools and use cases to model available qualifications and to check their compliance with respect to a given ASIL in a tool chain.

A more detailed description of the classes, attributes and relations can be found in the user manual of the Tool Chain Analyzer tool [TCA] that has been developed by Validas to support the tool chain modeling (see Fig 4) described here.

Building a formal model requires some effort, but it brings the following benefits:

- Models can be adapted easily, e.g. updating names in a consistent manner
- Models can be automatically checked for inconsistencies via plausibility checks:
    - Emptiness checks: Unchecked errors, use cases without errors, unused artifacts

    - Error assignment checks: Without dataflow from use case A to use case B via an artifact, the use case B cannot detect errors of A and A cannot prevent errors in B. This restricts the set of selectable checks/restrictions that can be assigned to an error.
    - Checks for sufficient descriptions: Missing descriptions of elements or missing explanations for high mitigation probabilities or the postulation that a tool has no impact.
    - Checks for existence of adequate qualifications: If in a model all elements with a TCL > 1 have qualifications with appropriate methods (as specified in the ISO 26262 tables) for the selected ASIL, then the tool chain is ISO 26262 compliant.
- Models for tools can be easily exchanged and reused, even if the checks and restrictions are distributed on other tools, for example a code generator model which requires some checks of a test tool can be exported including only the required checks.
- Review of tool chain models is supported by having automatically generated review checklists for tool chains (tool/use case-artifact matrix) and the tool confidence model (list of errors and their assigned counter-measures with TD).
- Rework and Change-tracking is assisted. The modeler can express the reasons for his/her decisions as comments and mark elements as assumption in order to make clear that these elements express proposals (to be accepted) and not facts (existing checks, restrictions) in the current development process. The modeler can also deactivate elements with the option to restore them later, which allows to analyze the effect of removing a tool to the TCLs of the remaining tools.
- Automatic generation of reports for model description (text+figures) and the derivation of the TCL for each use case and tool. The report is structured according to the ISO 26262 requirements for the work product *tool criteria evaluation report*.

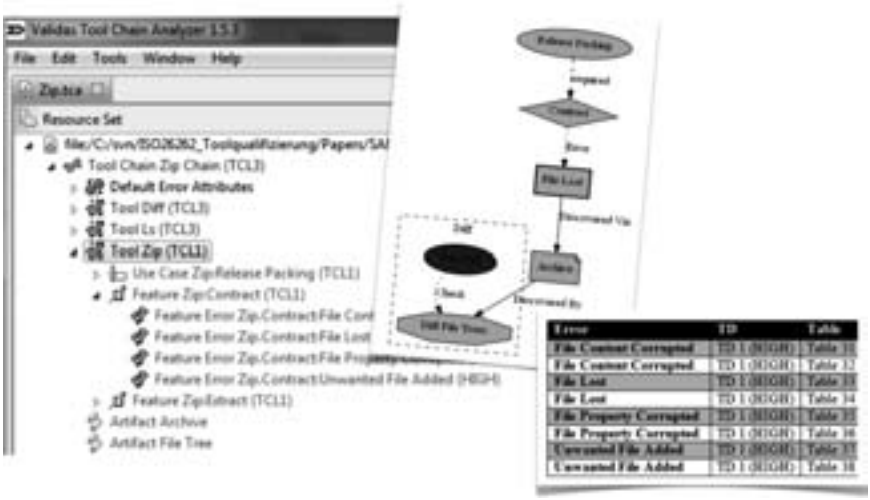


Fig 4. Tool Chain Analyzer with generated figures and tables (MS Word)

4 Tool Chain Extensions and Rearrangements

An important part of evaluating a tool chain is to find and assign checks or restrictions for the identified potential errors. There are various ways to rearrange or extend an existing tool chain (see Fig 5), such that it becomes more resilient to tool errors. Often introducing additional checks before or after a use case helps to detect or prevent errors. For example the error “Wrong Options Used” by the use case Compile-App from the example above can be detected by having a “Logfile Inspection” check in the use case. With “Logfile Inspection” we mean redirecting every compiler call to a log file and then asking the user to inspect this log file for correct compiler calls.
A very general counter measure is to introduce redundant and independently developed tools for the same purpose. For example many errors from the tool chain above can be detected by having a validation test run, where an alternative coverage tool and compiler is used, at least once before the product is released. In this context special care is required to avoid common cause failures. As an example, common cause failures may occur because of common components, e.g. commercial parser frontends for embedded compilers.
Also note that there are circumstances, where it suffices to have structural coverage measured in other tests on the same code. For example if 100% MC/DC is a test aim of two different test levels and different coverage tools are used at both levels one can argue that there is a high chance that missed paths in the product code that are not detected in one test bench due to “Wrong Coverage Reported” will be detected in the other with high probability provided the acceptance criteria are equally strong. However, an argumentation like this needs to be worked out carefully and requires checking side-conditions: For instance, are the same test design methods used, and are and equivalent test goals, e.g. full functional requirement coverage, applied in both test levels?

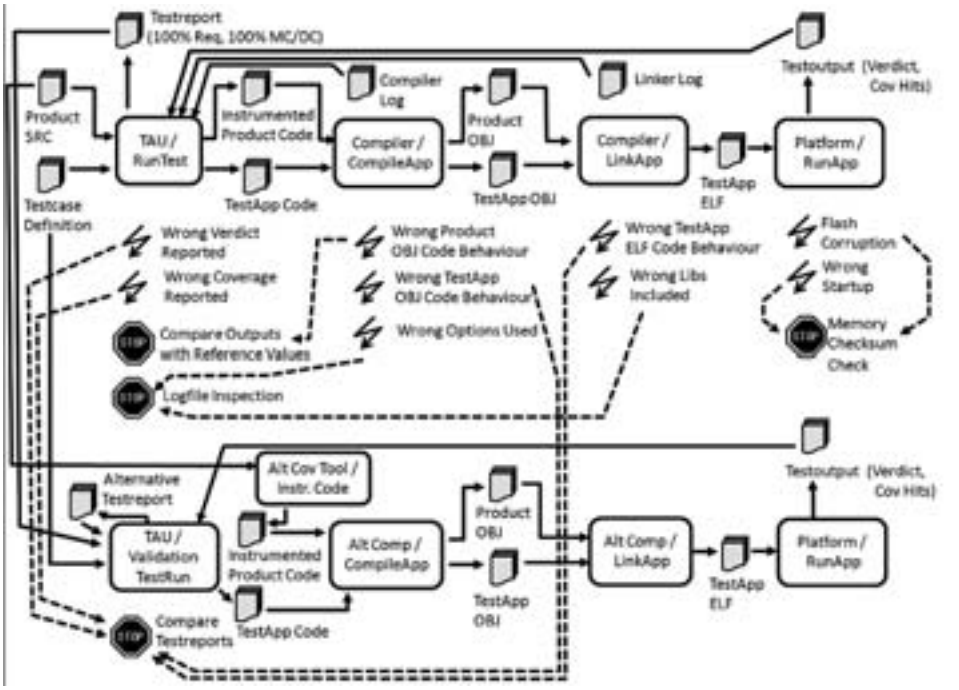


Fig 5. Extended testing tool chain, all errors have counter measures

5 Industrial Experience

We have applied the model based approach to a real industrial tool chain consisting of 37 tools, which is used for automotive SW development of hardware near SW modules. In total we have identified 136 distinct use cases in this tool chain over the whole development lifecycle, e.g. Requirements, Design, Coding, Unit Tests, Integration Tests, System Tests and Maintenance.

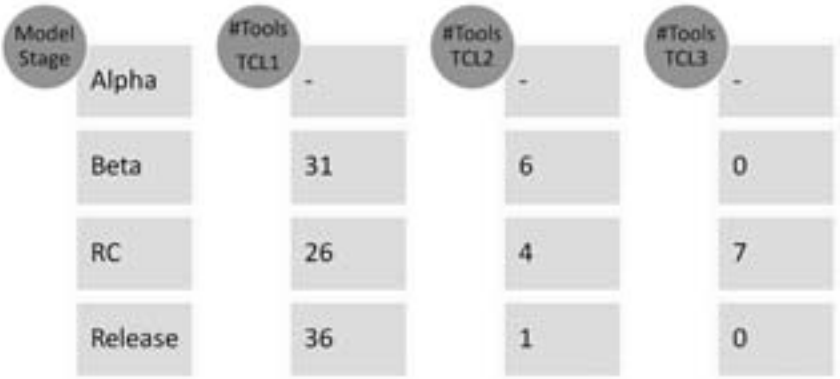


Fig 6. TCL status of the industrial tool chain during evaluation

The tool chain evaluation was carried out iteratively having the following phases Alpha, Beta, RC, Release. In each phase some adjustments have been made, e.g. dropping unfeasible checks or restrictions or considering some of these as they are implemented in the development process. As a consequence the TCL of tools varied from stage to stage as shown in the table above (see Fig 6).

The aim of the Alpha model was to get the tool chain structure complete and correct that is to have all tools, artifacts and use cases captured.

In the Beta model the tool chain analysis added potential errors and possible counter measures for these. In a detailed review of this Beta model the tool experts, that is the developers that use these tools in the development project, rejected, confirmed or added errors, checks or restrictions in the Beta model.

The RC model contained all potential errors and only checks and restrictions that are already implemented in the development process. The RC model thus reflects the tool confidence situation of the initial development process. In this initial process 11 tools have TCL2 or TCL3 and would thus require qualification.

In the Release model some extra checks and restrictions, which have been identified and accepted during evaluation have been added to the process. Most of these extra countermeasures were "quick wins", that is measures that do not require much extra effort, e.g. Logfile Inspection.

By implementing this reworked tool chain the TCL for 10 tools could be brought to TCL1, only one tool remained at TCL2. Hence, the effort spent in analyzing and reworking the tool chain paid off by saving the need for qualifying 10 tools. In addition the model made clear what functions of the remaining TCL2 tool need to be qualified and a suitable qualification kit can be constructed or demanded from the tool vendor.

The effort for the model-based tool chain evaluation varied between 0,5 and 5 days (2 days average) per tool. The effort increases with the number of use cases and interactions to other tools. A monolithic tool, e.g. Zip, is typically simple whereas distributed tools are typically complex, e.g. a test tool with lots of interactions to other tools (compiler, coverage tool, repository) and distributed processes (test application running on eval board, test management running on PC). Using a suggested default tool evaluation or classification from the literature [LPP10] with which we were able to derive the TCL for 25 of the 37 tools within one hour, resulted in qualification needs for 12 tools.

If we compare this with the result of only one tool to be qualified as achieved in our model and process based evaluation we get a big difference in terms of costs: assuming only 10[1] days of work for qualification per tool (or comparable amounts of money) this would cause 120 days of qualification work, which is the double amount of effort than classifying 25 tools with 2 days each and qualifying only one tool (25*2+10=60). Furthermore 12 tools cannot be classified using this default classification [LLP10] as they do not fall into one of the mentioned categories.

6 Conclusion

In this paper we have proposed a meta model for tool chains and a method for using this modeling for tool chain evaluation, that is TCL determination, according to ISO 26262. Having a model-based tool chain evaluation brings the following benefits:

- Automation (TCL computation, Report Generation)
- Adaptivity (Incremental Changes)
- Explorability (What effect do changes have?)
- Rework support (Review checklists, Change tracking)
- Enhanced quality assurance due to automated plausibility checks

The safety norm ISO 26262 offers a lot of flexibility to reach confidence in the use of tools. It considers tools together with the process they are used in and thus offers the possibility to adapt a process such that potential tool errors are detected with high probability within this process. This is a great opportunity to reduce the costs for tool qualification.
On the other hand the use of qualified tools can reduce the process overhead and optimize the development and test processes by eliminating some checks and restrictions needed for working with unqualified tools (even if they can be partially automated). Note that sometimes qualification kits that can be purchased bring along additional checks and restrictions, e.g. input range restrictions, to ensure the correct use of the tool in the perimeter of the use cases and functions covered by the qualification kit. Finding the optimal balance between development cost and qualification costs depends on the number and size of use cases of the tool and differs in each application scenario or company.The author area is a Word table. To insert or remove authors, insert or remove columns through Word's Table menu. Use this menu also to adjust table and column width.

[1] If qualification kits are not available it may be much more work to qualify a tool by building an adequate qualification kit.

Acknowledgements

This work has been supported by ARTEMIS and the German Federal Ministry of Research and Education (BMBF) within project RECOMP under research grant 1IS10001A. Furthermore we would like to thank Stefan Dirix for his contributions to the Tool Chain Analyzer.

References

[61508] International Electrotechnical Commission, IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systmes, Edition 2.0, Apr 2010.

[Be10] Beine M. A Model-Based Reference Workflow for the Development of Safety-Critical Software. In: Embedded Real Time Software and Systems, 2010.

[DO178] Radio Technical Commission for Aeronautics, RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification, Dec 13 2011

[DO330] Radio Technical Commission for Aeronautics, RTCA DO-330, Software Tool Qualification Considerations, Dec. 13 2011

[Hi11] Hillebrand J., Reichenpfader P, Mandic I., Siegl H., and Peer C. Establishing Confidence in the Usage of Software Tools in Context of ISO 26262. In SAFECOMP'11, 2011.

[L92] Laprie J.C. (ed.): Dependability: Basic Concepts and Terminology. Book. Springer-Verlag, 1992. ISBN 0-387-82296-8

[ISO26] International Organization for Standardization. ISO 26262 Road Vehicles –Functional safety–. 1st Edition, 2011-11-15

[LPP10] Löw P., Pabst P., Petry E., Funktionale Sicherheit in der Praxis: Anwendung von DIN EN 61508 und ISO/DIS 26262 bei der Entwicklung von Serienprodukten, 2010, dpunkt Verlag.

[SC04] Stürmer I. and Conrad M. Code Generator Certification: a Testsuite-Oriented Approach. In Proc of Automotive-Safety & Security, 2004.

[SS07] Schneider S., Slotosch O. A Validation Suite for Model-based Development Tools In CONQUEST 2007.

[TCA] Validas Tool Chain Analyzer. Obtainable freely from www.validas.de/TCA.html.

[WPS12] Wildmoser M., Philipps J., Slotosch O. Determining potential errors in tool chains – Strategies to reach tool confidence according to ISO26262 Submitted to SAFECOMP 2012.

Static Verification of Non-Functional Software Requirements in the ISO-26262

Daniel Kästner, Christian Ferdinand

AbsInt GmbH
Science Park 1
66123 Saarbrücken
kaestner@absint.com
ferdinand@absint.com

Abstract: The norm ISO-26262 aims at ascertaining the functional safety of Automotive Electric/Electronic Systems. It is not focused on purely functional system properties, but also demands to exclude nonfunctional safety hazards in case they are critical for a correct functioning of the system. Examples are violations of timing constraints in real-time software and software crashes due to runtime errors or stack overflows. The ISO-26262 ranks the static verification of program properties among the prominent goals of the software design and implementation phase. Static program analyzers are available that can prove the absence of certain non-functional programming errors, including those mentioned above. Static analyzers can be applied at different stages of the development process and can be used to complement or replace dynamic test methods. This article gives an overview of static program analysis techniques focusing on non-functional program properties, investigates the non-functional requirements of the ISO-26262 and discusses the role of static analyzers in the ISO-26262.

1 Introduction

The norm ISO-26262 [ISO11a] is based on the Functional Safety Standard IEC 61508 [IEC10] and aims at ascertaining the functional safety of Automotive Electric/Electronic Systems. It has been published as an international standard in August 2011, replacing the IEC 61508 as formal legal norm for road vehicles.

The ISO-26262 is not focused on purely functional system properties, but also demands to exclude nonfunctional safety hazards in case they are critical for the correct functioning of the system. Examples are violations of timing constraints in real-time software and software crashes due to runtime errors or stack overflows. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage. This also holds for related safety standards like DO-178B [Rad], DO-178C, IEC-61508 [IEC10], and EN-50128 [CEN09]. The ISO-26262 demands to consider the non-functional safety goals throughout the entire development process: from specification, architectural design and implementation to validation and testing.

While the ISO-26262 does not enforce specific testing and verification methods, the im-

portance of static verification is emphasized: it is considered one of the three goals of the software unit design and implementation stage. The advantage of static techniques is that they do not depend on specific test cases but provide results which are only based on the source (or binary) code. Testing, in general, cannot show the absence of errors [Rad] since only a concrete set of inputs is considered and usually no full test coverage can be achieved.

The term *static analysis* is used to describe a variety of program analysis techniques with the common property that the results are only based on the software structure. The most advanced static analysis technique is the so-called *Abstract Interpretation* [CC77] which is counted to the formal methods. Abstract Interpretation is a semantics-based methodology for program analysis. It provides results that are valid for all program runs with all inputs, thus enabling full control and data coverage. Abstract Interpretation-based tools are in industrial use that can prove the absence of stack overflows, compute safe upper bounds on the worst-case execution time [SLH+05], and prove the absence of runtime errors [DS07].

Specifically for non-functional program properties like timing, stack size, and runtime errors dynamic testing is difficult since identifying safe end-of-test criteria is an unsolved problem. In consequence the required test effort is high, the tests require access to the physical hardware and the results are not complete. Let's consider timing as an example: With modern processors the timing behavior of an instruction depends on the instructions previously executed, e.g., due to cache or pipeline effects. In consequence even MC/DC coverage is not enough to determine worst-case execution time information. This is because the code can be thoroughly tested in multiple runs without ever having exercised the longest path on any single run. Since different execution paths cannot be distinguished there is no control which paths have been covered. In contrast, Abstract Interpretation-based static analyses can be seen as equivalent to testing with full coverage. Thus, in the areas where validation by static analysis is technically feasible and applied in industry it defines the state-of-the-art testing technology. As an example the static WCET analyzer aiT [SLH+05] has been used by NASA in the Toyota Motor Corporation Unintended Acceleration Investigation to investigate the timing behavior, and could show the absence of timing-related software defects [NAS11].

In the following we will give an overview of the various different static analysis techniques focusing on Abstract Interpretation (Sec. 2). Sec. 3 discusses the predominant non-functional safety goals: timing correctness, absence of stack overflows, and absence of runtime errors. The application of static analysis techniques to demonstrate these goals is shortly sketched. The main part of the article is Sec. 4 which tracks the non-functional requirements through the different sections of Chap. 6 of ISO-26262 [ISO11b]. Sec. 5 discusses the integration of static analysis tools in the development process, focusing on interfaces, tool couplings, and tool qualification. Sec. 6 concludes.

2 Static Analysis Techniques

Static program analyzers compute information about the software under analysis without actually executing it. The analyzers can work at the source code level, or at the object or executable code level. The term "static analyzer" is very broadly used and can denote a variety of different techniques. This section gives an overview based on [CCF+07].

Sometimes the term "static analyzer" is understood as including style checkers looking for deviations from coding style rules (like Safer C checking for C fault and failure modes [Hat95] or MISRA C checker [Mot04]). Such syntactical methods are not "semantics-based", and thus, e.g., cannot determine variable values or check for correct runtime behavior.

Semantics-based static analyzers use an explicit (or implicit) program semantics that is a formal (or informal) model of the program executions in all possible or a set of possible execution environments. Based on the program semantics, information about data and control flow is obtained. The most important characteristics of static analyzers is whether they are *sound* or *unsound*. A static analyzer is called *sound* if the computed results hold for any possible program execution.

A program analyzer is unsound when it can omit to signal an error that can appear at runtime in some execution environment. Unsound analyzers are *bug hunters* or *bug finders* aiming at finding some of the bugs in a well-defined class. Their main defect is unreliability, being subject to false negatives thus claiming that they can no longer find any bug while many may be left in the considered class.

Unsoundness may be caused by

- discarding program effects:
  A simple example is a write access through a pointer, e.g. `*p=42`. If the pointer analysis is imprecise and concludes that `p` may point to 1000 variables then the only sound way to proceed is to assume that all 1000 variables are modified by this assignment and may take the value 42. Of course, this would make the result very imprecise. Hence, some unsound analyzers may decide to ignore instructions on which they know they probably are very imprecise. As an example the source code analyzer CMC [ECH+01] from Coverity ignores code with complex semantics whose analysis would be too hard to design. Some C compilers can produce stack usage estimates but they fail to take the effect of inline assembly code or certain library routines into account, thus possibly underestimating the maximal stack usage.

- ignoring some types of errors:
  The source code analyzer UNO concentrates exclusively on uninitialized variables, nil-pointer references, and out-of-bounds array indexing [Hol02]. Another example is bug pattern mining (cf. e.g., Klocwork K7™ [Klo]), looking for just a few key types of programmer errors. Due to high false alarm rates, also alarm filtering often is applied, e.g., Splint [LE01] sorts the most probable messages according to common programming practices.

- disregarding some runtime executions:
  Some static stack analyzers ignore recursions and only count the stack usage of one single execution of a recursive function. Dynamic analyzers, i.e., test & measurement approaches are unsound since it is impossible to explore all possible executions of a program on a machine.

- changing the program semantics:
  Source code analyzers have to take the machine representation of numbers into account. It is not correct to assume that integers are in $\mathbb{Z}$, and reals in $\mathbb{R}$. Instead the limited bitwidth of machine numbers in general, and especially floating-point rounding errors have to be taken into account.

2.1 Abstract Interpretation

The set of all possible runtime executions of a program is defined by the so-called concrete semantics. Unfortunately most interesting program properties are undecidable in the concrete semantics. The theory of abstract interpretation [CC77] offers a semantics-based methodology for static program analyses where the concrete semantics is mapped to a simpler abstract model, the so-called abstract semantics. The static analysis is computed with respect to that abstract semantics. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. By skilful definition of the abstract semantics a suitable trade-off between precision and efficiency can be obtained. Abstract interpretation supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an overapproximation of the concrete semantics. Moreover it can be shown that imprecisions always occur on the safe side. Examples of such proofs can be found in [Fer97, The04, Min04].

Let's illustrate this with two application scenarios: In runtime error analysis, soundness means that the analyzer never omits to signal an error that can appear in some execution environment. If no potential error is signalled, definitely no runtime error can occur, i.e., there are *no false negatives*. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm (*false positive*). This imprecision is on the safe side: it can never happen that there is a runtime error which is not reported. In WCET analysis, soundness means that the computed WCET bound holds for any possible program execution. Safety means that the only imprecision occurring is overestimation: the WCET must never be underestimated. Abstract Interpretation allows these soundness and safety properties to be formally proven.

3 Non-Functional Safety Goals

Safety standards like ISO-26262 [ISO11a], DO-178B [Rad], DO-178C, IEC-61508 [IEC10], and EN-50128 [CEN09] require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Some non-functional safety hazards can be critical for the correct functioning of the system: violations of timing constraints in real-time software and software crashes due to runtime errors or stack overflows. Proving their absence is explicitly listed as a verification goal by all of the above mentioned standards. In this chapter we will describe the background and the challenges of demonstrating the absence of these non-functional safety hazards.

3.1 Absence of Stack Overflow

A possible cause of catastrophic failure is a stack overflow which might cause the program to behave in a wrong way or to crash altogether. When they occur, stack overflows can be hard to diagnose and hard to reproduce. The problem is that the memory area for the stack usually must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for the selected set of program runs with fixed inputs. Even many repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed.

3.2 Worst-Case Execution/Response Time Guarantees

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Timing verification implies a reliable verification of code execution times, response times and end-to-end latencies to find the critical corner-case scenarios of system execution in the presence of complex processor architectures, multi-tasking operating systems, and network protocols. First, safe upper bounds for the execution times (WCETs) of non-interrupted tasks have to be determined. Then the worst-case response times (WCRTs) of an entire system from the task WCETs and information about possible interrupts and their priorities have to be computed [KFH+11]. In the following we will focus on determining the WCET of a task; for more information about system-level WCRT analysis see [HHJ+05].

Modern hardware makes computing safe and precise WCET bounds a challenge. There is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance microcontrollers and DSPs). Pipelines enable acceleration by overlapping the executions of

different instructions. Consequently the execution behavior of the instructions cannot be analyzed separately since it depends on the execution history. Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) real-time systems, this is undesirable and possibly even hazardous. Making the safe yet – for the most part – unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

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

Static analysis by Abstract Interpretation is a safe method for timing analysis providing guaranteed upper bounds for the worst-case execution time (WCET) of tasks [SLH+05]. Static analyzers are available for complex processors and yield precise results. However, a basic requirement is that the timing behavior of the processor is *predictable*. In general, predictability degrades with increasing interference by accesses to shared resources. In single-core processors this can be observed with speculative hardware mechanisms like caches, out-of-order pipelining, or branch prediction. If a task $T1$ on a processor with caches is preempted by another task, the memory blocks needed by $T1$ might be evicted from the cache. If $T1$ continues after the preemption additional cache misses can be the consequence. Covering all potential preemption scenarios by measurements clearly is infeasible. Cache-related preemption costs can be considered by static analysis, but the difference between average and worst-case execution time grows. By cache partitioning or cache locking predictability can be improved and the potential variance of execution time decreases.

On multi-core processors not only the interferences within each core have to be considered. Now there are additional interferences due to concurrent accesses to shared resources by different cores. Such interferences may be caused by accesses to common caches, common memory banks, or common FLASH or prefetch buffers. For a given multi-core architecture potential interferences have to be carefully examined. Then a configuration can be determined which enables or facilitates predictable performance [CFG+10].

3.3 Absence of Runtime Errors

Another important goal when developing critical software is to prove that no runtime errors can occur. Examples for runtime errors are division by zero, invalid pointer accesses, array bound violations, or arithmetic overflows. A well-known example for the possible effects of runtime errors is the explosion of the Ariane 5 rocket in 1996 [Lea96]. As detailed above, software testing can be used to detect errors, but not to prove their absence. E.g. to reliably detect a division by zero, all potential values of each variable of the denominator would have to be exerted.

The result of a sound static runtime error analysis for a statement x will be either "(i) statement x will not cause an error", or "(ii) statement x may cause an error". In the first case, the user can rely on the absence of errors, in the second case either an error has been found, or there was a false alarm (false positive). The imprecision induced by the semantical abstraction allows to compute results in acceptable time, even for large software projects. Nevertheless the results are reliable, i.e., the analysis will only err on the safe side: if the analyzer does not detect any error, the absence of errors has been proven - the coverage is 100%.

Each alarm the analyzer reports has to be manually investigated to determine whether there is an error which has to be corrected, or whether it was just a false alarm. If all the alarms raised by an analysis have been proved to be false, then the proof of absence of runtime errors is completed. This could be checked manually, but the problem is that such a human analysis is error-prone and time consuming, especially since there might be interdependencies between the false alarms and in some cases deviations from the C standard may be willingly accepted. Therefore the number of alarms should be reduced to zero, since then the absence of runtime errors is automatically proved by the analyzer run.

To that end, it is important that the analyzer is precise, i.e., produces only few false alarms. This can only be achieved by a tool that can be "specialized" to a class of properties for a family of programs. Additionally the analyzer must be parametric enough for the user to be able to fine-tune the analysis of any particular program of the family. General software tools not amenable to specialization usually report a large number of false alarms which is the reason why such tools are only used in industry to detect runtime errors, and not to prove their absence.

Additionally the analyzer must provide flexible annotation mechanisms to communicate external knowledge to the analyzer. Only by a combination of high analyzer precision and support for semantic annotations the goal of *zero false alarms* can be achieved. A prerequisite is that users get enough information to understand the cause of an alarm so that they can either fix the bugs or supply the missing semantic information, e.g., about sensor input ranges.

3.4 Tool Support

The absence of non-functional errors from the categories listed above can be formally proven by Abstract Interpretation based static analysis tools. Examples are aiT [SLH+05] for worst-case execution time analysis, StackAnalyzer [FHF07] for stack usage analysis and Astrée [DS07] for runtime error analysis. For a more comprehensive overview of static analysis tools see [WEE+08] and [CCF+07].

Since runtime error analysis targets unspecified behaviors which are not covered by the language semantics the static analysis necessarily has to be based on the source code.

In contrast, for safe WCET and stack usage analysis it is important to work on fully linked *binary code*, i.e., here the static analysis is not based on the source code but on the executable code. In general, neither a sound stack usage analysis nor a sound WCET analysis

can be based on the source code. The compiler has a significant leeway to generate machine code from the source code given which can result in significant variations of stack usage and execution time. Moreover, for precise results it is important to perform a whole-program analysis after the linking stage. Lastly, for timing analysis precisely analyzing cache and pipeline behavior is imperative to obtain precise time bounds. This is only possible when analyzing machine instructions with known addresses of memory accesses.

Over the last few years, a more or less standard architecture for timing analysis tools has emerged [Erm03, FHL+01]. It neither requires code instrumentation nor debug information and is composed of three major building blocks:

- control-flow reconstruction and static analyses for control and data flow,
- micro-architectural analysis, computing upper bounds on execution times of basic blocks,
- path analysis, computing the longest execution paths through the whole program.

The data flow analysis of the first block also detects infeasible paths, i.e., program points that cannot occur in any real execution. This reduces the complexity of the following micro-architectural analysis. There, basic block timings are determined using an abstract processor model (*timing model*) to analyze how instructions pass through the pipeline taking cache-hit or cache-miss information into account. This model defines a cycle-level abstract semantics for each instruction's execution yielding in a certain set of final system states. After the analysis of one instruction has been finished, these states are used as start states in the analysis of the successor instruction(s). Here, the timing model introduces non-determinism that leads to multiple possible execution paths in the analyzed program. The pipeline analysis has to examine all of these paths. In general, the availability of safe worst-case execution time bounds depends on the predictability of the execution platform. Especially multi-core architectures may exhibit poor predictability because of essentially non-deterministic interferences on shared resources which can cause high variations in execution time. [CFG+10] gives a more detailed overview and suggests example configurations for available multi-cores to support static timing analysis.

This analysis architecture can also be applied to stack usage analysis. While for timing analysis the entire machine state has to be modelled, for stack usage analysis only stack pointer manipulations have to be covered.

4 Non-Functional Software Requirements in the ISO-26262

In this section we focus on [ISO11b], which specifies the requirements for product development at the software level for automotive applications. It covers the typical software development phases: requirements for initiation of software development, specification of software safety requirements, software architectural design, software unit design and implementation, software unit testing, software integration and testing, and verification of

software safety requirements. Non-functional software safety requirements are addressed in all these stages along the V-Model.

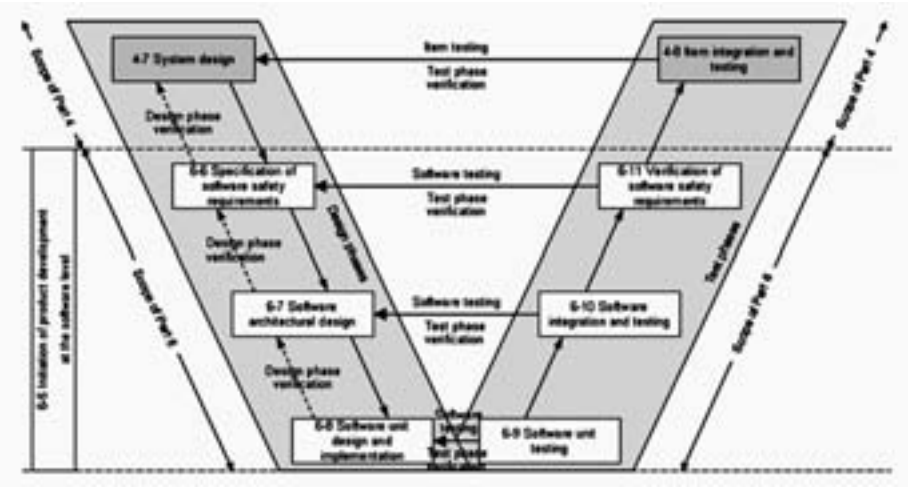


Figure 1: V-Model in the ISO-26262

4.1 Basic Requirements

The ability to handle non-functional program properties is one determining factor for selecting a suitable modelling or programming language (cf. [ISO11b], Sec. 5.4.6). The standard requires that real-time software and runtime error handling must be supported. It also states that ”criteria that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines or by the development environment”. Table 1 of [ISO11b] suggests the use of language subsets to exclude language constructs which could result in unhandled runtime errors. However, for typical embedded programming languages like C or C++, runtime errors can be caused by any pointer or array access, by arithmetic computations, etc. So in this case the absence of runtime errors has to be ensured by appropriate tools as a part of the development environment. Also timing and stack behavior is not captured in current programming language semantics and has to be addressed by specific tools.

4.2 Safety Requirements

In general the specification of the software safety requirements considers constraints of the hardware and the impact of these constraints on the software. Among others, safety requirements apply to functions that enable the system to achieve or maintain a safe state, and

to functions related to performance or time-critical operations. The standard explicitly lists some requirements which are part of the software safety, including the hardware-software interface specification, the relevant requirements of the hardware design specification, and the timing constraints. Hardware- or configuration-related errors (e.g., stack overflows and runtime errors like erroneous pointer manipulations) can cause globally unpredictable behavior affecting all safety functions and thus have to be taken into account. Timing constraints include the response time at the system level with derived timing properties like the worst-case execution time.

4.3 Software Architectural Design

The architectural design has to be able to realize the software safety requirements. The requirements listed in [ISO11b] include verifiability, feasibility for the design and implementation of the software units, and the testability of the software architecture during integration testing. In other words, the *predictability* of the system is one of the basic design criteria since predictability of the software as executed on the selected hardware is a precondition both for verifiability and for testability. Sec. 7.4.17 of [ISO11b] explicitly demands that "an upper estimation of required resources for the embedded software shall be made, including the execution time, and the storage space". Thus, upper bounds of the worst-case execution time and upper bounds of the stack usage are a fixed part of the architectural safety requirements. The importance of timing is is also reflected by the fact that "appropriate scheduling properties" are highly recommended for all ASIL levels as a principle for software architecture design. All existing schedulability algorithms assume upper bounds on the worst-case execution time to be known, as well as interferences on task switches either to be precluded or predictable. Thus the availability of safe worst-case execution and response times belong to the most basic scheduling properties.

Software architectural design requirements also explicitly address the interaction of software components. "Each software component shall be developed in compliance with the hightest ASIL of any requirements allocated to it." Furthermore, "all of the embedded software shall be treated in accordance with the highest ASIL, unless the software components meet the criteria for coexistence [...]" (cf. [ISO11b], Sec. 7.4.9.-7.4.10). Freedom of interference is an essential criterion for coexistence. Here all considerations of Sec. 3.2 apply. Freedom of interference also is addressed by Annex D of [ISO11b]. It discusses timing properties like worst-case execution time or scheduling characteristics as well as memory constraints. For memory safety, corruption of content, as well as read or write accesses to memory allocated to another software elements have to be excluded. Such accesses can be caused by stack overflows or runtime errors like erroneous pointer manipulations and dereferences. As a technique to show the absence of memory faults, Annex D lists static analysis of memory accessing software.

Table 6 of [ISO11b] lists the methods for verification of the software architectural design. Control flow analysis and data flow analysis are recommended for ASIL-A and ASIL-B and highly recommended for ASIL-C and ASIL-D; formal verification is recommended for ASIL-C and ASIL-D. This can be done separately for the modelling and the imple-

mentation level. However, with model-based code generation there is a synergy between model-level and implementation-level analysis (cf. Sec. 4.4). Since the semantics of the model is given by the generated implementation, source or binary code analyses can be used to verify the architectural design by propagating analysis results from the implementation level to the modelling level. This is well supported by static analysis techniques (cf. Sec. 5).

4.4 Software Unit Design, Implementation, and Testing

Chap. 8 of [ISO11b] defines the three goals of software unit design and implementation as: specification of the software units in accordance with design and safety requirements, implementation of the software units, and static verification of design and implementation of the software units. Thus, static verification plays a very prominent role in the design and implementation stage; it should always precede dynamic testing which should focus on properties not statically verified.

Table 9 lists the methods for verification of software unit design and implementation, including formal verification, control flow analysis, data flow analysis, static code analysis and semantic code analysis. As detailed in Sec. 2 all these techniques can be considered as aspects of general static analysis. Data and control flow analysis is a natural part of semantics-based static analyzers. With sound static analyzers proofs of data and control flow behavior can be obtained; they can be counted to the formal methods. A sound static analyzer for runtime error analysis like Astrée also provides safe data and control flow analysis without additional effort. The mentioned static analysis techniques are (highly) recommended for all ASIL levels.

In the structure of the ISO-26262 there is a differentiation between implementation, addressed in Chap. 8, and testing, addressed in Chap. 9. The absence of runtime errors (division by zero, control and data flow errors, etc) is considered as a robustness property (Sec. 8.4.4) which has to be ensured during implementation. Resource usage contraints like timing or stack consumption are addressed in Chap. 9 of [ISO11b] (Software Unit Testing). This distinction corresponds to the boundary between source code and binary code; apparently the underlying assumption is that source code can be analyzed and binary code has to be tested, which does not account for static analyses working at the binary code level. The requirements become consistent again when static analysis techniques are counted to the testing methods. Actually static analysis can be seen as an exhaustive testing method providing full coverage – a view shared by related safety standards like DO-178B [Rad], IEC-61508 [IEC10], and EN-50128 [CEN09].

4.5 Software Integration and Testing

The software integration phase has to consider functional dependences and the dependences between software integration and hardware-software integration. Again the non-

functional software properties have to be addressed; robustness has to be demonstrated which includes the absence of runtime errors, and it has to be demonstrated that there are sufficient resources to support the functionality which includes timing and stack usage (Sec. 10.4.3). Like in the unit testing stage, the ISO-26262 points out typical limitations of dynamic test and measuring techniques: when dynamic tests are made the test coverage has to be taken into account and it has to be shown that code modification and instrumentation does not affect the test results (cf. Sec. 9.4.5.-9.4.6; 10.4.5-10.4.6). It is important to perform WCET analysis in the unit testing stage to get early feedback of the timing behavior of the software components. However, it has to be noted that since the WCET depends on the memory addresses in the program and can be influenced by linking decisions, it also has to be applied in the integration stage on the final executable. The same considerations apply to interactions or interferences between software components.

5 Static Analysis Tools in the Development Process

Static analysis tools compute information about the software under analysis without actually executing it. No testing on physical hardware is required. As a consequence, static analyzers can be conveniently integrated in automatic build or testing systems in a fashion similar to the compilers.

Furthermore, it is straightforward to couple static code analysis tools with other development tools, e.g., with model-based code generators. After generating the code from the model, the static analyses can be automatically invoked from the modelling level. Also their results can be made available at the modelling level. Examples for such tool couplings exist between the static analyzers aiT, StackAnalyzer, Astrée and model-based code generators like dspace TargetLink, Esterel SCADE [ET], or ETAS ASCET. Industry partners of the European FP7 project INTERESTED where a complete integrated timing tool chain has been developed report reductions in overall project effort between 20% and over 50% [Int11].

Also tool couplings between different timing analysis tools can be beneficial. One example is a tool coupling between the aiT WCET Analyzer and the scheduling analysis tool SymTA/S [HHJ+05]. Here the goal is to provide a seamless framework for timing analysis at the code and system level providing both worst-case execution time and worst-case response time bounds. The advantage of such tool couplings is that they make it easy to use the static analysis during software development and not only in the final validation and verification stage. Defects can be detected early, so that late-stage integration problems can be avoided.

Many safety standards, including the ISO-26262, require development tools to be qualified, according to the criticality level of the tool usage. The qualification of static analysis tools can be automated to a large degree by dedicated *Qualification Support Kits* (QSKs). They consist of a report package and a test package. The report package lists all functional requirements and contains a verification test plan describing one or more test cases to check each functional requirement. The test package contains an extensible set of test

cases and a scripting system to automatically execute all test cases and evaluate the results. The generated reports can be submitted to the certification authority as part of the certification package. Additional Qualification Support Life Cycle Data (QSLCD) reports may be necessary to demonstrate the compliance of the tool development process to the standards for safety critical software development. QSKs and QSLCDs enable successful tool qualification for all safety standards up to the highest tool confidence levels.

6 Summary

The norm ISO-26262 requires to identify functional and non-functional safety hazards and to demonstrate that the software does not violate the relevant safety goals. Critical non-functional safety hazards can be caused by violations of timing constraints in real-time software and software malfunctioning due to runtime errors or stack overflows.

For such non-functional properties dynamic test and measurement techniques are often inappropriate since it is very hard to determine a safe test end criterion and typically no full test coverage can be obtained. One technique which achieves full control and data coverage and produces results valid for all program runs with all inputs is Abstract Interpretation, a semantics-based methodology for static program analysis. The term static analysis is used to describe a variety of program analysis techniques. We have given an overview of the different categories of static analyzers and outlined the principles of static analyzers for non-functional program properties. Abstract Interpretation-based tools are in industrial use that can prove the absence of stack overflows, runtime errors, and which can compute safe upper bounds on the worst-case execution time. The qualification of these tools as required by safety standards like the ISO-26262 can be automated to a large extend by dedicated Qualification Support Kits.

The ISO-26262 ranks the static verification of program properties among the prominent goals of the software development process. We have detailed the requirements of the ISO-26262 for non-functional program properties and discussed the role of static analyzers in the ISO-26262.

Acknowledgements

The work presented in this paper has been supported by the European FP6 project INTEREST, the European FP7 projects INTERESTED and PREDATOR, the ITEA2 project ES_PASS, and the ARTEMIS project MBAT.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[CCF+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of Static Analyzers: A Comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007*, pages 3–20. IEEE Computer Society, 2007.

[CEN09] CENELEC DRAFT prEN 50128. Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, 2009.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:26–42, 2010.

[DS07] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, pages 437–451, 2007.

[ECH+01] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *18th ACM Symp. on Operating Systems Principles*, 2001.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[ET] Esterel Technologies. SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite`.

[Fer97] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[FHF07] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static Memory and Timing Analysis of Embedded Systems Code. In Perry Groot, editor, *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS 2007) on March 23, 2007 at Eindhoven, The Netherlands*, volume 07-04 of *TUE Computer Science Reports*, pages 153–163, Eindhoven, 2007.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[Hat95] L. Hatton. Safer C: Developing for High-Integrity and Safety-Critical Systems. *McGraw-Hill*, 1995.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis – the SymTA/S Approach. *IEEE Proceedings on Computers and Digital Techniques*, 152(2), March 2005.

[Hol02] G. Holzmann. UNO: Static Source Code Checking for User- Defined Properties. In *6th World Conf. on Integrated Design and Process Technology (IDPT02)*, 2002.

[IEC10] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[Int11] INTERESTED Project Final Report (Public Version). `http://www.interested-ip.eu/`, 2011.

[ISO11a] ISO/FDIS 26262. Road vehicles – Functional safety, 2011.

[ISO11b] ISO/FDIS 26262. Road vehicles – Functional safety – Part 6: Product development at the software level, 2011.

[KFH+11] D. Kästner, C. Ferdinand, R. Heckmann, M. Jersak, and P. Gliwa. An Integrated Timing Analysis Methodology for Real-Time Systems. *Embedded World Congress*, 2011.

[Klo] Klocwork. Klocwork K7^{TM}. `http://www.klocwork.com`.

[LE01] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *2001 USENIX Security Symposium, Washington, D.C.*, 2001.

[Lea96] J.L. Lions et al. ARIANE 5, Flight 501 Failure. *Report by the Inquiry*, 1996.

[Min04] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[Mot04] The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*, October 2004. ISBN 0-9524156-2-3.

[NAS11] NASA Engineering and Safety Center. Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation, 2011.

[Rad] Radio Technical Commission for Aeronautics. RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification.

[SLH+05] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[The04] Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, , and Per Stenstroem. The Determination of Worst-Case Execution Times —Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

Extraktion von Interthread-Kommunikation in eingebetteten Systemen

Martin Wittiger Steffen Keul

Universität Stuttgart, Institut für Softwaretechnologie
Universitätsstr. 38, 70569 Stuttgart

{martin.wittiger, steffen.keul}@informatik.uni-stuttgart.de

Abstract: Mit der zunehmenden Verbreitung von Multicore-Rechnern werden Multicore-Architekturen auch in eingebetteten Systemen mehr und mehr Einzug halten. Zusätzlich zu den Schwierigkeiten der Softwareentwicklung für Singlecore-Plattformen müssen Software-Ingenieure somit die Herausforderungen bewältigen, bestehende Systeme zuverlässig und fehlerfrei auf Multicores zu portieren und dabei dennoch das Parallelisierungspotential möglichst effektiv zu nutzen. Bislang existiert kaum Werkzeugunterstützung, um diese Portierung in der Praxis durchzuführen. Unsere Arbeit verfolgt das Ziel, Algorithmen und Werkzeuge zu entwickeln, die existierende Steuersoftware im Automotive-Bereich semi-automatisiert auf Multicore-Plattformen portieren können. In diesem Beitrag wird eine statische Analysetechnik vorgestellt, mit der aus dem Quelltext eines eingebetteten Systems Kommunikationsgraphen extrahiert werden können. Diese können verwendet werden, um Modifikationsbedarf in bestehender Software zu identifizieren, und eignen sich als Grundlage für die spätere Partitionierung. Die vorgestellten Algorithmen wurden prototypisch in unserer Programmanalyse-Toolsuite Bauhaus implementiert und ihre prinzipielle Tauglichkeit wurde durch Anwendung auf bestehende industrielle Softwaresysteme bestätigt.

1 Einleitung

Moderne Anwendungen in eingebetteten Systemen, wie sie unter anderem im Automotive-Bereich eingesetzt werden, benötigen erhebliche Rechenleistung. Sie stehen auf der einen Seite dem beständigen Anwenderwunsch nach neuer und noch komfortablerer Funktionalität gegenüber, müssen jedoch auf der anderen Seite harte Anforderungen an Antwortzeiten erfüllen. Aufgrund der zunehmenden Verfügbarkeit von Multicore- und der zu erwartenden abnehmenden Verfügbarkeit leistungsfähiger Singlecore-Plattformen besteht der Wunsch, existierende Systeme auf Multicore-Plattformen lauffähig zu machen.

Daher soll nun Regelungssoftware, die für Singlecore-Systeme entwickelt wurde, auf Multicores portiert werden. Eine konservative Strategie mit geringem Portierungsaufwand besteht offensichtlich darin, die Software nur auf einem Kern auszuführen und weitere Kerne

Diese Arbeit wurde im Rahmen des Projektes ARAMiS durch das Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01IS11035 gefördert. Die Verantwortung für den Inhalt liegt bei den Autoren.

ungenutzt zu lassen. Um aber rechenintensivere Probleme zu bearbeiten, sollte natürlich eine echte Parallelisierung angestrebt werden. Die Parallelisierung von vormals sequentiellem Programmcode birgt Risiken für die Zuverlässigkeit des Systems. In nebenläufigem Code können Race-Conditions auftreten, die fatales Fehlverhalten der Software bewirken können. Für eine Portierung müssen zu parallelisierende Einheiten identifiziert werden und es muss, z. B. durch statische Programmanalyse, nachgewiesen werden, dass diese Einheiten frei von Race-Conditions parallelisierbar sind [BSER11], oder es müssen entsprechende Modifikationen vorgenommen werden.

Ist eine Entscheidung getroffen, welche Programmteile parallel ausgeführt werden können, so muss im nächsten Schritt eine möglichst optimale Partitionierung, also die Verteilung von Threads auf Kerne, gefunden werden. Wir gehen davon aus, dass eine statische Zuweisung von Threads an Kerne der CPU erfolgen soll. Eine Partitionierung der Software weist also jedem Thread einen Kern zu, auf dem ihm Laufzeit zur Verfügung gestellt wird. Die Migration von Threads, wie sie bei dynamischem Scheduling erfolgt, ist ausgeschlossen, da ein solcher Systemaufbau nicht dem Stand der Technik zur Entwicklung sicherheitskritischer eingebetteter Systeme entspricht [RJFG10].

Die Qualität einer Partitionierung lässt sich anhand verschiedener Eigenschaften bemessen. Eine Partitionierung muss die Kerne der CPU möglichst gleichmäßig belasten – was auch beinhaltet, keinen Kern überzubelegen –, um Tasks die Möglichkeit zu geben, ihre Deadlines einzuhalten, und um kurze Antwortzeiten zu ermöglichen. Eine gleichmäßige Belastung vereinfacht auch eine spätere Erweiterung des Systems.

Wir gehen davon aus, dass Kern-interne Kommunikation Kern-übergreifender Kommunikation vorzuziehen ist. Als Vorteile Kern-interner Kommunikation konnten wir identifizieren:

- Ist bekannt, dass bestimmte Tasks auf dem gleichen Kern laufen werden, so hat dies bei der Synchronisation zwischen ihnen verlaufender Kommunikation Vorteile. Solche Kern-interne Kommunikation kann möglicherweise durch sonst nur Singlecore-Systemen vorbehaltene Synchronisationsmuster (zum Beispiel Deaktivierung von Interrupts) abgesichert werden, oder es kann sogar ganz auf explizite Synchronisation verzichtet werden, weil gegenseitiger Ausschluss schon aus dem Nebenläufigkeitsmodell hergeleitet werden kann. Dabei sinkt der Portierungsaufwand, und Leistungseinbußen durch Synchronisation werden vermieden [SKS10].
- In AUTOSAR 4.0 ist der `SetResource`-/`GetResource`-Mechanismus nur für die Synchronisation auf einem einzigen Kern einsetzbar, sodass zur Synchronisation von gemeinsamen Speicherbereichen zwischen verschiedenen Kernen nur Spin-Locks eingesetzt werden können. Diese bedingen jedoch ein andauerndes wiederholtes Prüfen der Verfügbarkeit des Spin-Locks (Busy-Waiting) und können somit eine geringere Auslastung der einzelnen Kerne erzwingen.
- Auch vermuten wir, dass Software, die vorwiegend Kern-interne Kommunikation verwendet, besser von Caches profitieren kann – insbesondere bei Schreibzugriffen.

Aufgrund dieser Annahme ist in unserem Modell Kommunikation zwischen Kernen generell unerwünscht. Anders ausgedrückt: Greifen unterschiedliche Threads auf dieselben

globalen Variablen zu, ist es wünschenswert, dass diese Threads auf dem gleichen Kern laufen.

Darüber hinaus kann es weitere Einschränkungen geben, denen eine Partitionierung entsprechen soll. So kann der Entwickler aufgrund (lizenz-)rechtlicher, technischer oder architektonischer Gegebenheiten die Anforderung haben, dass bestimmte Threads einem spezifischen Kern oder zumindest dem gleichen Kern zugeordnet werden. Denkbar sind auch gegensätzliche Einschränkungen: Tasks von unterschiedlichen Zulieferern sollen auf keinen Fall auf dem gleichen Kern ausgeführt werden.

2 Analyse

Wir gehen im Folgenden davon aus, dass der Benutzer die gewünschte Parallelität in einem eingebetteten System planen und potentiell parallele Einheiten identifizieren kann. Die Spezifikation der gewünschten Parallelität kann durch geeignete Mittel direkt im Quelltext des zu analysierenden Systems oder durch eine separate Konfigurationsdatei erfolgen. Hierzu müssen die Startfunktionen der einzelnen Threads angegeben werden.

Als konkreter Thread sollen alle Programmteile gelten, die parallel zueinander ausgeführt werden können. Zur Extraktion von Abhängigkeiten zwischen nebenläufigen Programmeinheiten führen wir den Begriff eines abstrakten Threads ein. Ein abstrakter Thread ist die Repräsentation von konkreten Threads in der statischen Programmanalyse. Die Unterscheidung zwischen konkreten und abstrakten Threads kann von Bedeutung sein, falls in einem Programm zur Laufzeit neue Threads erzeugt werden können und möglicherweise die Anzahl der konkreten Threads nicht präzise abgeschätzt werden kann. Als abstrakter Thread gilt also:

- der Boot-Loader, d. h. der Programmteil, der zum Start des eingebetteten Systems mit der Ausführung beginnt und evtl. das Betriebssystem startet,
- Interrupt-Handler, also Funktionen des Programms, die als Reaktion auf externe Ereignisse aufgerufen werden, inkl. aller davon aufgerufenen Funktionen,
- durch das Betriebssystem verwaltete Programmeinheiten (in OSEK/VDX OS „Task“ genannt),
- evtl. durch API-Funktionen wie z. B. der POSIX-Funktion `pthread_create` dynamisch gestartete Threads. In unserem Modell gelten alle auf diese Weise aus einer Programmstelle erzeugten Laufzeit-Threads als ein abstrakter Thread. Diese Variante wird in eingebetteten Systemen typischerweise nicht verwendet.

Unser Programmanalyse-Modell identifiziert die abstrakten Threads in der Software und erzeugt für jeden einen Aufrufgraphen. Im Aufrufgraphen existiert jede Funktion als Knoten und jeder Funktionsaufruf als Kante von der aufrufenden zur aufgerufenen Funktion.

Um den Aufrufgraphen aufzubauen, müssen daher Informationen über die möglichen Ziele von Zeigern in dem Programm vorliegen, um indirekte Funktionsaufrufe durch Funkti-

onszeiger aufzulösen. Hierbei verlangen wir, dass die eingesetzten Analysen konservativ sind. Die Zeigeranalyse ist konservativ, falls sie garantiert alle möglichen Zeigerziele berechnet. Dabei dürfen aber die Zielmengen überschätzt und Zeigerziele annotiert werden, die in keiner realen Ausführung des Programms tatsächliche Ziele sein können. Diese Konservativität ist notwendig, da fast alle Fragestellungen der Programmanalyse im Allgemeinen unentscheidbar sind. Unsere Zeigeranalyse orientiert sich an der fluss- und kontextinsensitiven Analyse nach Pearce et al. [PKH04].

Unsere Analysewerkzeuge sind als Teil der Programmanalyse-Suite Bauhaus [RVP06] implementiert. Unsere Analysealgorithmen werden auf eine abstrakte Programmzwischendarstellung angewendet, sodass eine beliebige Programmiersprache als Quelle für die Analyse dienen kann. Aufgrund der weiten Verbreitung der Programmiersprache C im Bereich eingebetteter Systeme beschränken wir unsere Beschreibungen auf diese Quellsprache.

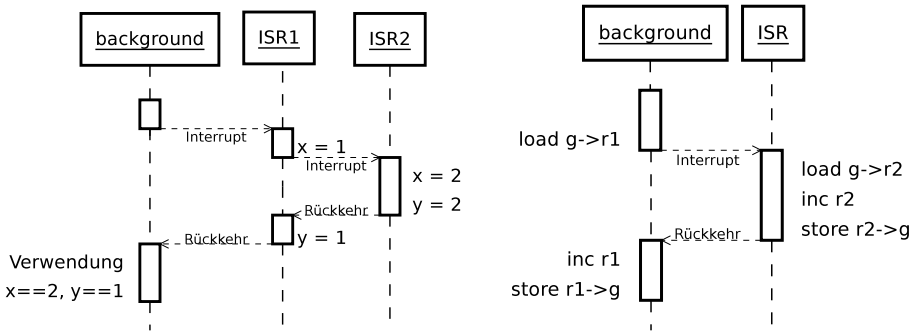
Nachdem die gewünschte Parallelisierung festgelegt ist und die Zwischendarstellung für das Programm erzeugt wurde, sollen nun die Analysen wie folgt angewendet werden. Zunächst wird das nebenläufige Programm mit einer Race-Condition-Analyse auf gravierende Kommunikationsfehler hin untersucht, wie in Abschnitt 2.1 beschrieben. Sind keine Fehler (mehr) vorhanden, erfolgt der Aufbau des in Abschnitt 2.2 beschriebenen Zugriffsgraphen. Der dazu verwendete Extraktionsalgorithmus wird in Abschnitt 2.3 präsentiert.

2.1 Data-Race-Analyse

Die grundlegenden Probleme, die durch Race-Conditions entstehen, sind verlorene Aktualisierung und das Lesen inkonsistenter Datenstrukturen. In Abbildung 1(a) werden inkonsistente Verwendungen illustriert. Zwei Interrupt-Handler, `ISR1` und `ISR2`, aktualisieren zwei Variablen, `x`, `y`. Durch die Verzahnung der einzelnen Zuweisungen werden im Hintergrund-Task jeweils die Aktualisierung aus `ISR1` an `y` und aus `ISR2` an `x` sichtbar. Existiert zwischen den beiden Variablen eine funktionale Abhängigkeit, so liegt eine Inkonsistenz vor, die zu einer Fehlfunktion des Systems führen kann. In Abbildung 1(b) hingegen wird parallel in dem Interrupt-Handler `ISR` und in dem Hintergrund-Task `background` eine Inkrementierung der globalen Variable `g` ausgeführt. Dazu wird der gerade aktuelle Wert von `g` jeweils in ein Register (`r1` oder `r2`) geladen und inkrementiert. Dann wird das Resultat wieder in `g` gespeichert. Durch die Verzahnung der einzelnen Anweisungen geht die Aktualisierung durch den `ISR` verloren und nur eine Inkrementierung der Variablen hat einen Effekt, was in vielen Fällen nicht die beabsichtigte Semantik darstellt.

Ein Data-Race ist eine Situation, in der mehrere Threads ohne Synchronisation auf eine Speicherstelle zugreifen und mindestens einer dieser Threads die Speicherstelle verändert. Die Data-Races sind eine Teilmenge der Race-Conditions. Es existieren einige Programmanalysatoren, die in der Lage sind, Data-Races zu identifizieren [PFH06, NAW06, VJL07]. Es existieren auch Forschungsarbeiten, die gezielt Data-Race-Analysatoren für den Automotive-Bereich bei Einsatz eines OSEK/VDX OS untersuchen [Keu11, SSV+11].

Während Data-Races als einzelne unzureichend synchronisierte Speicherzugriffe definiert sind, können bei einer Portierung auch komplexere Race-Conditions auftreten, deren Feh-



(a) inkonsistente Verwendung (b) verlorene Aktualisierung

Abbildung 1: Beispiele für Auswirkungen von Race Conditions

lerpotential nicht einfach erkennbar ist. Es existieren jedoch zumindest Ansätze zur Erkennung von Higher-Level Data-Races [AHB03, RFP11] und zur Identifikation von unerwünschtem Datenfluss über Thread-Grenzen hinweg [Keu10].

Auf der Zwischendarstellung des Programms kann eine Race-Condition-Analyse durchgeführt werden, um die Korrektheit der gewünschten Parallelisierung zu prüfen oder gegebenenfalls Korrekturen am Quelltext vorzunehmen, um die Korrektheit herzustellen.

2.2 Zugriffsgraphen

Um die Aufgabe der Partitionierung eines Systems zu beschreiben, führen wir als neue Datenstruktur den Zugriffsgraphen ein. Dieser Graph soll die Kopplung zwischen den Threads beschreiben. Er kann automatisiert in den Kommunikationsgraphen (siehe Abschnitt 3) transformiert werden.

Der Zugriffsgraph enthält zwei verschiedene Arten von Knoten, abstrakte Threads und Shared Variables. Als Shared Variable bezeichnen wir alle Speicherbereiche, die potentiell von mehr als einem konkreten Thread verwendet werden. Als Verwendung gilt das Lesen oder das Schreiben des Werts.

Der Zugriffsgraph ist bipartit. Für jedes Paar eines abstrakten Threads und einer Shared Variable wird eine Kante erzeugt, falls der abstrakte Thread auf diese Shared Variable zugreift. Weitere Kanten existieren nicht.

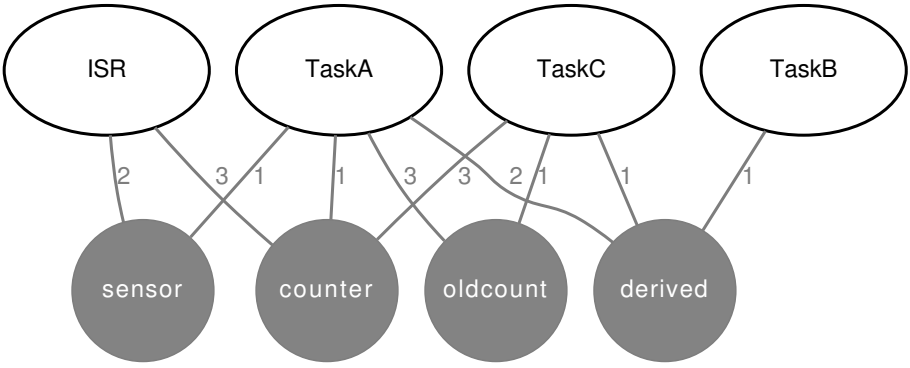
Kanten werden mit einem Gewicht annotiert, das die Kopplung einer Shared Variable an einen Thread beschreibt. Ein großes Gewicht soll die Threads, die auf eine gemeinsame Shared Variable zugreifen, in einer nachfolgenden Partitionierung tendenziell auf einen gemeinsamen Kern zusammenziehen.

Als mögliche Bewertungen für das Gewicht einer Kante (T, V) zwischen einem abstrakten Thread T und einer Shared Variable V haben wir identifiziert:

```
1  int counter = 0;
2  int oldcount = -1;
3  int sensor = 0;
4  int derived = 0;
5
6  void ISR()
7  {  sensor = read();
8     counter++;
9  }
10
11 void TaskA()
12 {  if (oldcount < counter) {
13        derived = work(sensor);
14        oldcount = counter;
15 }  }
```

```
1  void TaskB()
2  {
3     log(derived);
4  }
5
6  void TaskC()
7  {
8     int reset;
9     if (oldcount == counter) {
10       reset = process(derived);
11       if (reset) {
12          counter = 0;
13       }
14    }
15 }
```

(a) Beispielquelltext



(b) Zugriffsgraph

Abbildung 2: Beispiel für die Extraktion eines Zugriffsgraphen

```
1  DeclareResource(M);
2  int global;
3
4  void Task_A()
5  {  GetResource(M);
6     if (global > 7) {
7        ...
8     }
9     ReleaseResource(M);
10 }
```

```
11 void TaskB()
12 {
13    while (...) {
14       GetResource(M);
15       current = global;
16       ...
17       global = current;
18       ReleaseResource(M);
19 }  }
```

Abbildung 3: Beispiel für Zugriffe mit konsistenter Sperrung

1. Ein konstantes Gewicht c_r, falls T mindestens einen lesenden Zugriff auf V enthält, ein konstantes Gewicht c_w, falls T einen schreibenden Zugriff auf V enthält, und $c_r + c_w$, falls beide Arten in T enthalten sind.

2. Die Anzahl der Zugriffsstellen im Programmcode, an denen T auf V zugreift.

3. Um die tatsächliche Häufigkeit eines Zugriffs stärker zu gewichten, kann zusätzlich die Tiefe der Schachtelung in Schleifen berücksichtigt werden. Liegt in diesem Fall ein Zugriff auf eine Shared Variable innerhalb einer Funktion, die aus einer Schleife heraus aufgerufen wird, so wird dieser Zugriff stärker bewertet als ein Zugriff aus rein sequentiellem Code.

4. Aus ähnlicher Motivation heraus kann auch die Zugriffshäufigkeit eines konkreten Threads T auf V in bestimmten Testausführungen der Software durch dynamische Analyse ermittelt und als eine Metrik verwendet werden.

5. Eine Metrik, die stärker auf Wartezeiten der Threads fokussiert, gewichtet Zugriffe auf Semaphoren V besonders, indem der Kante (T, V) das Gewicht der Zugriffe auf alle Shared Variables innerhalb der kritischen Region zugewiesen wird. Diese Metrik zielt darauf ab, kritische Regionen möglichst demselben Kern zuzuteilen. In diesem Fall kann, je nach Nebenläufigkeits-Modell, dadurch die kritische Region sogar überflüssig werden und in sequentiellen Code ohne Synchronisation übersetzt werden.

6. Sind alle Zugriffe auf eine oder mehrere Shared Variables stets durch Sperrung einer Semaphore konsistent in allen Threads geschützt, so kann die Gewichtung für die Shared Variables ausschließlich der Kante zu der Semaphore zugeschlagen werden. Dies kann in unserer Analysekette einfach durch Anwendung der klassischen Lockset-Analyse [Keu11] erreicht werden. In Abbildung 3 werden Zugriffe auf die Semaphore `M` durch Kanten mit besonders hohem Gewicht im Zugriffsgraphen repräsentiert, und die Zugriffe auf die Shared Variable `global` verstärken das Gewicht dieser Kanten.

Ein Beispiel für einen Zugriffsgraphen wird in Abbildung 2 gezeigt. In dem Beispielquelltext sind vier Funktionen definiert, `ISR`, `TaskA`, `TaskB` und `TaskC`, die separate

parallelisierbare Prozesse einer Steuersoftware darstellen sollen. Sie greifen auf gemeinsame globale Variablen zu. In dem Zugriffsgraphen werden die Threads als helle, ovale Knoten angezeigt und die Shared Variables als dunkle, kreisförmige Knoten. Durch die Kanten wird angezeigt, welche Zugriffe tatsächlich durchgeführt werden. Für die Kantengewichtung haben wir eine sehr einfache Strategie gewählt: Zu Beginn ist das Gewicht der Kante 0. Gibt es mindestens einen lesenden Zugriff, wird das Gewicht der Kante um eins erhöht. Analog dazu wird, wenn es mindestens einen schreibenden Zugriff gibt, das Gewicht um 2 erhöht. Es werden nur Kanten mit Gewicht > 0 angelegt. Somit ist z. B. erkennbar, dass im Beispiel der Thread `TaskB` lesend auf die Shared Variable `derived` zugreift (in Zeile 10), oder dass der Thread `TaskA` sowohl lesend als auch schreibend auf die Shared Variable `oldcount` zugreift (in den Zeilen 12 und 14).

2.3 Extraktionsalgorithmus

Zum Aufbau der Zugriffsgraphen werden die Aufrufgraphen der abstrakten Threads nacheinander durchlaufen. Innerhalb einer Funktion werden die Zugriffsstellen mit Zugriffen auf Shared Variables einzeln betrachtet und gegebenenfalls eine Kante in den Zugriffsgraphen eingefügt. Das Resultat der Zeiger-Analyse wird verwendet, um die Ziel-Variablen an Dereferenzierungen zu bestimmen.

Zur Ermittlung des Kantengewichts werden entsprechende Annotationen an den Kanten vorgenommen. Die Gewichtungsfunktion ist parametrierbar ausgeprägt, um verschiedene Varianten evaluieren zu können.

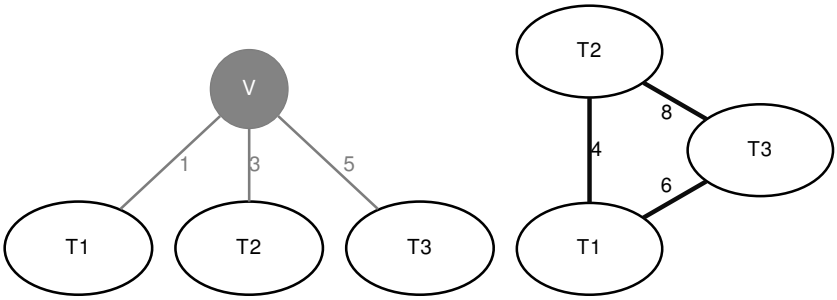
In unserer prototypischen Implementierung wird Strategie 1 aus Abschnitt 2.2 umgesetzt: Gewicht 1 wird für die Existenz eines lesenden Zugriffs und Gewicht 2 für die Existenz eines schreibenden Zugriffs annotiert.

3 Kommunikationsgraphen

Als Datenstruktur zur Partitionierung eines Systems definieren wir den Kommunikationsgraphen. Er kann aus dem in Abschnitt 2.2 beschriebenen Zugriffsgraphen abgeleitet werden.

Um den Zugriffsgraphen in den Kommunikationsgraphen umzuwandeln, müssen die Shared Variables aufgelöst und durch Kommunikationskanten zwischen Threads ersetzt werden.

Eine Shared Variable, die über n Kanten mit den Gewichten $w_1 \ \ldots \ w_n$ mit n Threads verbunden ist, wird zu $m = n(n-1)/2$ Kommunikationskanten zwischen den n Threads. Die Gewichte dieser m Kanten muss der Algorithmus berechnen. Hier sind verschiedene Berechnungswege möglich. Eine einfache Umsetzung weist der Kommunikationskante zwischen zwei Threads die Summe der Zugriffskantengewichte zu. Diese Variante lässt sich wie folgt beschreiben:



(a) Ausgangssituation (Zugriffsgraph) (b) Resultat (Kommunikationsgraph)

Abbildung 4: Auflösung einer Variablen mit Zugriffen aus drei Threads

Hat eine Shared Variable V Zugriffskanten zu den Threads $T_1, \ldots, T_n$, wobei $\forall i.\, 1 \leq i \leq n$ die Kante $(V,\ T_i)$ das Gewicht w_i hat, fügt der Algorithmus dem Graphen $\forall i, j.\, 1 \leq i < j \leq n$ eine Kommunikationskante $(T_i,\ T_j)$ mit Gewicht $w_i + w_j$ hinzu. Ist eine Kommunikationskante, die hinzugefügt werden soll, bereits vorhanden, wird das Gewicht der bestehenden Kante einfach entsprechend erhöht.

Abbildung 4 zeigt die Auflösung einer Variablen nach diesem Schema, auf die von $n = 3$ Threads aus zugegriffen wird.

Verschiedene Varianten dieser Heuristik sind anwendbar. Statt der Summe könnte auch das Maximum der beiden Werte verwendet werden. Auch können die Gewichte der Zugriffskanten, die die Variable mit anderen Threads verbinden, in die Berechnung mit eingehen. So könnte beispielsweise die Summe aller Zugriffskantengewichte multipliziert mit einem geeigneten Faktor in die Berechnung mit einfließen, um zu berücksichtigen, dass auch wenige Zugriffe auf eine hoch frequentierte Variable erheblichen Einfluss haben können.

Kommunizieren zwei Threads über mehr als eine Variable miteinander, so existiert im Grundalgorithmus trotzdem nur eine Kommunikationskante zwischen den Threads. Ihr Gewicht entspricht dann der Summe der Gewichte, die sich aus der Auflösung aller im Zugriffsgraphen benachbarten Variablen ergeben.

Der in Abbildung 5 dargestellte Kommunikationsgraph entsteht aus dem in Abbildung 2(b) dargestellten Zugriffsgraphen durch Anwendung des im obigen Beispiel beschriebenen Algorithmus. So ergibt sich beispielsweise das Gewicht der Kante zwischen den Threads `ISR` und `TaskA` aus der Summe der Kommunikation über die Variablen `sensor` ($2+1 = 3$) und `counter` ($3 + 1 = 4$) – zusammen also 7.

Zusätzlich wurde in der Abbildung eine Abschätzung der CPU-Utilisation der einzelnen Tasks annotiert. Diese muss vom Benutzer eingegeben werden und kann beispielsweise aus WCET-Werkzeugen übernommen werden oder aus empirischen Tests der Software stammen.

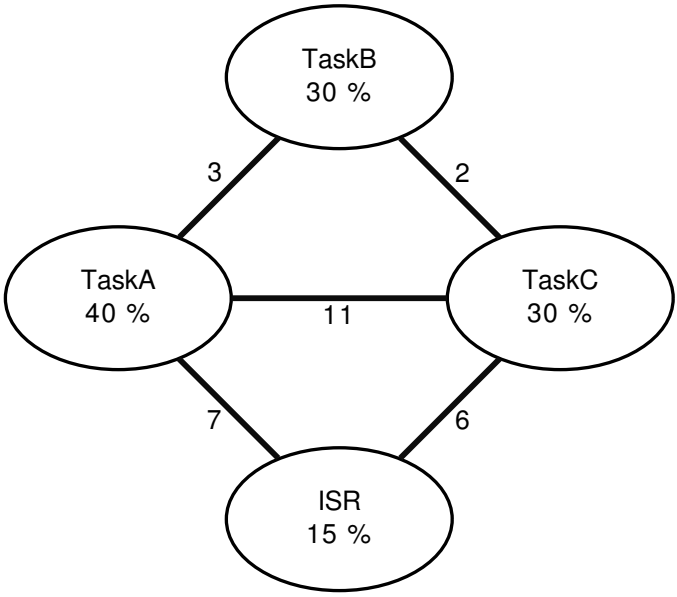


Abbildung 5: Kommunikationsgraph, aus dem Zugriffsgraphen in Abbildung 2(b) hergeleitet

4 Anwendungen des Kommunikationsgraphen

Der Kommunikationsgraph eines eingebetteten Systems stellt eine abstrakte Beschreibung des Datenaustauschs zwischen den einzelnen Threads dar.

4.1 Architekturdokumentation

Der Kommunikationsgraph kann eingesetzt werden, um die Kopplung zwischen Threads explizit zu dokumentieren und die Einhaltung von Architektur-Vorgaben zu prüfen. Ist vorgegeben, dass zwei Threads in einem System stets vollständig unabhängig voneinander agieren sollen, so kann die Einhaltung dieser Vorgabe durch das Fehlen einer Kante zwischen diesen Threads verifiziert werden.

4.2 Ermittlung von Synchronisationsbedarf

Steht nur eine Hypothese für eine möglicherweise geeignete Parallelisierung eines Systems zur Verfügung, so kann von diesem hypothetischen System ein Kommunikationsgraph erzeugt werden. Anhand des Graphen kann evaluiert werden, ob in dem hypothetischen System alle Kommunikationswege berücksichtigt wurden und insbesondere, ob die

Änderungen an der Singlecore-Implementierung vollständig durchgeführt wurden.

Zusätzlich kann die Möglichkeit geschaffen werden, die Kanten des Kommunikationsgraphen mit den Quelltext-Positionen zu annotieren, die diese Kanten hervorrufen. Dadurch wird es möglich, iterativ Hypothesen für geeignete Parallelisierungen zu testen und somit die Kosten einer Änderung des Systems gegenüber dem erwarteten Gewinn durch eine Parallelisierung abzuwägen.

4.3 Partitionierung

Eine weitere Verwendungsmöglichkeit für Kommunikationsgraphen ergibt sich bei der Partitionierung. Wir haben ein Werkzeug entwickelt, das basierend auf dem Kommunikationsgraphen die Software partitioniert. Hierbei beschränken wir uns auf die beiden am zentralsten erscheinenden Kriterien. Neben dem Kommunikationsgraphen wird eine Abschätzung der CPU-Utilisation der einzelnen Threads in die Software eingegeben. Das Partitionierungsproblem besteht für uns demnach darin, einen guten Kompromiss zwischen Gleichbelastung der Kerne und Minimierung der Interkern-Kommunikation zu finden. Im Idealfall werden alle Kerne gleich belastet und es findet keine Interkern-Kommunikation statt.

Für unsere prototypische Implementierung haben wir eine Bewertungsfunktion gefunden, die diese beiden Kriterien berücksichtigt. Sie lässt sich durch Parameter beeinflussen. So kann beispielsweise vom Nutzer vorgegeben werden, wie wichtig die Gleichbelastung der Kerne im Vergleich zur Minimierung der Kommunikation ist. Die Bewertungsfunktion weist jeder möglichen Partitionierung einen Wert zu, der umso niedriger ist, je besser die Partitionierung den Bewertungskriterien entspricht.

Es bleibt nun eine, gemessen an der Bewertungsfunktion näherungsweise optimale, Partitionierung zu finden. Hierbei handelt sich um ein numerisches Optimierungsproblem. Dieses lässt sich näherungsweise mit Simulated Annealing [KGV83, SK06] lösen. Unsere Implementierung kann damit Graphen, die aus industriell eingesetzter, eingebetteter Steuerungssoftware resultieren, auf üblichen Entwicklungssystemen in realistischer Zeit bearbeiten. Die berechnete Lösung wird dem Entwickler als Partitionierung vorgeschlagen.

5 Zusammenfassung und Ausblick

Wir haben in dem vorliegenden Beitrag eine Strategie skizziert, wie eine Migration sicherheitskritischer Software von Singlecore- zu Multicore-Plattformen durchgeführt werden kann. Nach einer Auswahl parallel ausführbarer Threads werden die Entwickler durch Programmanalyse bei ihrer Portierungsaufgabe unterstützt. Mittels Race-Condition-Analyse kann die Abwesenheit von Synchronisationsfehlern in Software validiert werden. Durch Extraktion des Kommunikationsgraphen wird eine automatisierte Generierung geeigneter Partitionierungen ermöglicht. Darüber hinaus haben wir mit dem Kommunikationsgraphen

eine Datenstruktur entworfen, die weitere wichtige Funktionen, besonders in der Architektur-Dokumentation und -Prüfung, übernehmen kann.

Als Fortsetzung dieser Arbeit wollen wir uns mit der Frage befassen, wie eine optimale Partitionierung erreicht werden kann. Wir vermuten, dass durch eine geeignete Wahl der Heuristiken zur Berechnung der Kantengewichte im Zugriffsgraphen bereits sehr gute Partitionierungen möglich werden.

Wir erwarten auch, dass unsere Technik durch iteriertes Prüfen von Parallelisierungs-Hypothesen wertvolle Hinweise zur Verteilung der einzelnen Tasks und Interrupt-Handler auf Threads geben kann.

Literatur

[AHB03] Cyrille Artho, Klaus Havelund und Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, November 2003.

[BSER11] Michael Bohn, Jörn Schneider, Christian Eltges und Robert Röger. Migration von AUTOSAR-basierten Echtzeitanwendungen auf Multicore-Systeme. In Fachgruppen des Fachbereichs Softwaretechnik der Gesellschaft für Informatik e.V., Hrsg., *Softwaretechnik-Trends*, Band 31 / 3, Seiten 9–14. Köllen Druck & Verlag GmbH, Bonn, 2011.

[Keu10] Steffen Keul. Static Versioning of Global State for Race Condition Detection. In *Ada-Europe '10*, Band 6106 der *LNCS*, Seiten 111–124. Springer, 2010.

[Keu11] Steffen Keul. Tuning Static Data Race Analysis for Automotive Control Software. In *SCAM '11*, Seiten 45–54. IEEE Computer Society, 2011.

[KGV83] Scott Kirkpatrick, C. Daniel Gelatt und Mario P. Vecchi. Optimization by simulated annealing. *Science*, (220):671–680, 1983.

[NAW06] Mayur Naik, Alex Aiken und John Whaley. Effective Static Race Detection for Java. In *PLDI '06*, Seiten 308–319, New York, NY, USA, 2006. ACM.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster und Michael Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI '06*, Seiten 320–331, New York, NY, USA, 2006. ACM.

[PKH04] David J. Pearce, Paul H. J. Kelly und Chris Hankin. Efficient field-sensitive pointer analysis for C. In *PASTE '04*, Seiten 37–42, New York, NY, USA, 2004. ACM.

[RFP11] Aoun Raza, Stefan Franke und Erhard Plödereder. Detecting High-Level Synchronization Errors in Parallel Programs. In Alexander Romanovsky und Tullio Vardanega, Hrsg., *Reliable Software Technologies – Ada-Europe 2011*, Band 6652 der *LNCS*, Seiten 17–30, Heidelberg, 2011. Springer.

[RJFG10] Kai Richter, Marek Jersak, Christian Ferdinand und Peter Gliwa. Eine ganzheitliche Methodik für den automatisierten Echtzeit-Nachweis zur Absicherung hoch integrierter, sicherheitskritischer Software-Systeme. In Hubert B. Keller, Erhard Plödereder, Peter Dencker und Herbert Klenk, Hrsg., *Automotive - Safety & Security 2010*, Seiten 27–41, Aachen, 2010. Shaker.

[RVP06] Aoun Raza, Gunther Vogel und Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Ada-Europe 2006*, Band 4006 der *LNCS*, Seiten 71–82. Springer, 2006.

[SK06] Johannes Josef Schneider und Scott Kirkpatrick. *Stochastic optimization*. Springer, Berlin, 2006.

[SKS10] Kathrin D. Scheidenmann, Michael Knapp und Claus Stellwag. Load Balancing in AUTOSAR-Multicore-Systemen. *Elektroniknet*, (3/2010):22–25, 2010.

[SSV+11] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich und Markus Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *POPL '11*, Seiten 93–104, New York, NY, USA, 2011. ACM.

[VJL07] Jan Wen Voung, Ranjit Jhala und Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC-FSE '07*, Seiten 205–214, New York, NY, USA, 2007. ACM.

MEMICS - Memory Interval Constraint Solving of (concurrent) Machine Code

Dirk Nowotka[1], Johannes Traub[2]

[1]Department of Computer Science, Kiel University,
dn@informatik.uni-kiehl.de
[2]E/E- and Software-Technologies, Daimler AG,
johannes.traub@daimler.com

Abstract: Runtime errors occurring sporadically in automotive control units are often hard to detect. A common reason for such errors are critical race conditions. The introduction of multicore hardware enables software to be run in parallel, and hence, drastically increases the vulnerability to such errors. Race conditions are difficult to discover by testing or monitoring, only. Hence, a static analysis of code is required to effectively reduce the occurrence of such errors. In this paper we introduce a new Bounded Model Checking tool, which in its core is an Interval Constraint Solver, operating on a machine code based model and is able to handle memory instructions directly. As control units are usually running on task-based operating systems like AUTOSAR or OSEK, our tool features a task model, which is able to handle sequential and concurrent task scheduling.

1 Introduction

Every control unit used in the automotive environment, e.g. driver assistance systems, is running on software systems and it is obvious that the reliability, safety, and security of such systems is of utmost importance. Possible runtime errors software can suffer from are e.g. arithmetic overflows, division by zero, NULL pointer dereferences, race conditions, stack overflows. A detailed list of runtime errors is shown in Section 3 in Table 1. Especially race conditions are hard to detect even in software running on singlecore hardware, however when it comes to multicore hardware it is far more difficult. A common task in the verification of software is the search for such runtime errors. The complexity of this problem is quickly growing with the complexity of the software itself and made much worse when parallel hardware is involved.

In contrast to common static analysis tools like Astrée [CCF+05] and Polyspace [pol], which analyse the source code via Abstract Interpretation [CC77a] and may suffer from potential false positives, this paper is focused on the formal verification of software. One of the techniques used in formal verification of software is Bounded Model Checking (BMC) [BCC+03]. In BMC a model, which in this case is the structural representation of the software, is unrolled up to some fixed depth and passed to a standard proof engine. Such a proof engine is e.g. a SAT-/SMT-Solver, which is used to solve the satisfiability (SAT)

[Coo71] problem or the satisfiability modulo theories (SMT) [dMB09] problem, like in CBMC [CKL04], F-Soft [IYG+04], and LLBMC [SFM10]. The main challenges that we address with the proposed tool are

efficient model unrolling which does directly affect the size of the formula,

internal handling of memory access of the code to be verified, and

implementation of a task model which provides the possibility to check software implementing a task-based operating system like AUTOSAR [Con] and OSEK [ose].

In the following, we elaborate on the aforementioned points and after that conclude the introduction by pointing out where our tool MEMICS advances the state of the art.

1.1 Model Unrolling

In Bounded Model Checking there exist various ways of unrolling the model. The two major approaches of unrolling are: external and internal. In the external variant the model is first unrolled into a logic formula and then passed to an independent proof engine. In each run the engine is started with an entirely new formula. Hence, it cannot reuse assignments already learnt from previous runs. In case of the internal variant the proof engine is directly embedded into the Model Checking tool. This allows for reusing already learnt clauses from previous runs achieved by conflict analysis during the search process. A Model Checker supporting internal unrolling is HySAT [FHT+07].

In [Tra10] we came up with the approach to use HySAT as proof engine in order to take advantage of the Bounded Model Checker and its underlying Interval Constraint Solver. Our approach was working fine for small problems. However, with the problem size exceeding around a hundred lines of code, HySAT was not able to handle the resulting model any more. We located the bottleneck inside the internal unrolling procedure of HySAT. In each iteration every variable of the HySAT model is newly initialized with its defined interval, unless it is not forced to keep its value of the current iteration. Due to that reason each variable representing a memory location must be encoded to store its value as long as the location is not part of a write operation in the current iteration. Therefore, the model and respectively the resulting formula are quickly growing to a prohibitively large size.

1.2 Interval Constraint Solving

Interval Constraint Solving (ICS) is a technique used to find a solution for a SMT input problem, but compared to a SMT-Solver ICS is operating on variable ranges. The search procedure of ICS, which is used in HySAT, is very similar to the common search algorithm used in standard SAT-/SMT-Solvers. An example of such a search algorithm based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62] is given in the Fig. 1. The main difference between ICS and this algorithm resides within the two functions

`decide()` and `bcp()`. Instead of computing a new variable decision in `decide()` the ICS computes new interval bounds for the variable to decide. A default splitting strategy for the interval $[min, max]$ can be $[min, \frac{max+min}{2}]$ and $(\frac{max+min}{2}, max]$. The function `bcp()` - abbreviating "boolean constraint propagation" - serves as an in place substitute for "interval constraint propagation" (`icp`) in ICS. In `icp` the current interval decision gets propagated through all the clauses containing the affected variable. If there exists a clause, which is false under the current interval decision, `icp` returns this clause as the conflict clause. If this conflict can not be resolved the input formula is unsatisfiable. If no conflicting clause exists the search procedure continues.

```
while (true) {
  if (!decide())
    return SAT;
  while (!bcp()) {
    if (!resolveConflict())
      return UNSAT;
  }
}
```

Figure 1: Basic Search Algorithm of a SAT-Solver from Chaff [MMZ+01].

1.3 Memory Access

Memory Access is a big challenge in BMC of software, because it has to be encoded into a logical formula. The memory access instruction for a n-byte integer with a common memory representation where an address has a size of one byte may have the following logic encoding:

$$\texttt{load:} \quad result_0 = memory[address] \wedge \Big(\bigwedge_{i=1}^{n} result_i = result_{i-1} \mid (memory[address + i] << (i * 8)) \Big)$$

$$\texttt{store:} \quad \bigwedge_{i=0}^{n} memory[address + i] = \big((value >> (i * 8))\&0\text{xff}\big)$$

Every memory access, which depends on a variable with a type-size of greater than one byte, results in at least two clauses in the logic formula. If the variable itself is a pointer, it can point to m different memory addresses, which results in m times bytesize(variable) clauses. Hence, the logic encoding of a function with memory access and even containing some pointers can quickly reach an enormous size.

1.4 Task Model

The introduction of a task model requires a proper scheduling mechanism. Scheduling in task-based operating systems like AUTOSAR and OSEK is based on different priorities per task and whether a task is preemptive or not. Also, interrupt service routines (ISRs) have to be considered, as an ISR is able to interrupt every task. To provide a proper logic encoding of such a task model and its scheduler is a big challenge. Especially the unrolling procedure has to be efficient, since all possible interleavings between tasks and ISRs have to be considered, which might quickly lead to a state explosion in the logic encoding.

In the present paper we introduce a new BMC tool consisting of an interval constraint solver, which

- handles memory access internally,
- supports floating point variables,
- is capable of handling sequential and concurrent task-based systems, and
- can reuse already learnt clauses from conflict analysis of previous iterations.

The introduction of this tool is located in Section 2. Section 3 shows some results of our implementation from several test runs on a set of testcases. Finally, we conclude our paper in Section 4 and also discuss potential future work.

2 The MEMICS tool

The MEMICS Model Checker operates based on the architecture shown in Fig. 2. First, the C/C++ source code is compiled into the LLVM Intermediate Representation (IR) [Lat] using the Clang [Fan10] compiler front-end of LLVM [LA04]. Second, the MEMICS model is created by invoking the LLVM code generator on the LLVM IR. The actual MEMICS process starts with unrolling the model into a logic encoding in Static Single Assignment [CC77b] form, which results in a set of clauses. If this set of clauses is empty, the model is safe. If not, the solve process is started. In case the solver does not find a solution for the current state, the next iteration step is computed. If the current state is satisfiable, the solver encountered a runtime error and has a valid counterexample.

The LLVM IR still features data structures like arrays and pointers. In addition the LLVM environment allocates virtual memory for each local variable in a function. Instead of providing a new lowering mechanism, when it comes to the transformation into a proper logic encoding, we decided to reuse the one already provided inside the LLVM Code Generator. As the LLVM IR is based on a MIPS like instruction set, we chose the MIPS Assembly language as a base for the MEMICS model. Hence using the LLVM Backend and the resulting MIPS Assembly Language to generate our model has the following advantages:

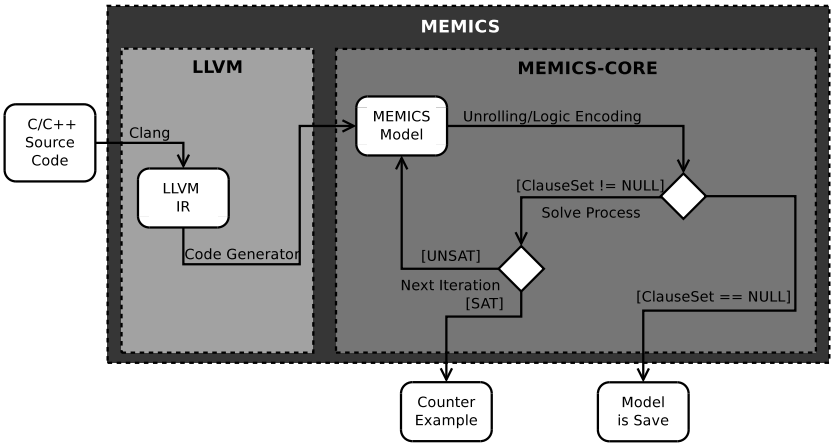


Figure 2: An Overview of the MEMICS Architecture.

- no dereferenciation of pointers,
- no derivation of Phi functions, and
- no assurance of memory allocation for local variables.

The only memory instructions, which we need to deduce, are those for heap based memory management: malloc and free.

2.1 The MEMICS Model

The model, the ICS operates on, is the state transition system M, consisting of a set of states $S = \{s_1, \ldots, s_n\}$ (the instructions) and a set of transitions $T = \{t_1, \ldots, t_m\}$. A transition t_i is defined as the 4-tuple: $< s_{i'}, a, c, s_{i''} >$. Where $s_{i'}$ is the current state or instruction, a the actual action to execute, c an obligatory condition to take the transition, and finally $s_{i''}$ the successor state. The instruction set our model is operating on, is derived from the MIPS Assembly Language [Swe06]. In order to properly handle this language our model also features a representation of a MIPS-CPU. Therefore the model has 32 variables representing the 32-bit registers, a stack pointer register and as well a program counter register in order to traverse over the transition system. Our model supports as many processor cores as required, where each core has the same amount of internal registers including a separate program counter register as well as a separate stack pointer register. The model also contains a vector of byte sized intervals representing the memory. The access to the memory, which is part of the solver itself, is described in Section 2.2.

2.1.1 Task Model

The task model is a set of tasks $R = \{r_1, \ldots, r_p\}$, where each task r_j consists of a subset of states $S' = \{s_{j_1}, \ldots, s_{j_k}\} \subseteq S$ of the MEMICS model M and a subset of transitions $T' = \{t_{j'_1}, \ldots, t_{j'_\ell}\} \subseteq T$. A transition t_i of a task is defined as the following 11-tuple:

$$< s_{i'}, a, c, s_{i''}, idle, run, start, wait, preempt, prio, freq >,$$

where $s_{i'}$, a, c, and $s_{i''}$ are defined as in the MEMICS model. The other components are required to provide a proper scheduling mechanism, where $idle$, run, $start$, and $wait$ represent the current status of the task, $preempt$ states whether it is preemptive or not, $prio$ defines it priority and $freq$ is the frequency, which the task is started with. The MEMICS model has also been equipped with a clock clk, in order to define the scheduling. The logic encoding of a non-conditional instruction is for example defined as:

$$(s_{i'} \wedge run \rightarrow a \wedge s_{i''}) \wedge (s_{i'} \wedge !run \wedge wait \rightarrow s_{i'})$$

The first implication is used to execute the current instruction of a task, if the task is running, otherwise the second implication takes place, which keeps the tasks waiting. In Fig. 3 the scheduling of a task is illustrated. The task can only be in one of the states: idle, start, run, wait at once.If a task is idle and the rule $clk \mod (freq - 2) == 0$ holds the start flag of the task is enabled. From start the task will either be set to run or wait, according to its priority. Finally if a task finished it will be set to idle again. Please note, that the state idle is not implemented into the logic, hence it can be represented as: $!run \wedge !wait \wedge !start$.

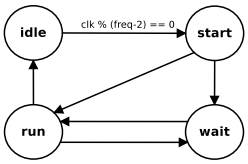


Figure 3: Scheduling of task

For a proper scheduling of all tasks on each core the following rules must hold, where the priority of t_i is higher than the priority of t_k:

1. $\sum_{j=1}^{n} run_j \leq 1$
2. $start_i \wedge \bigwedge_{f=1}^{i-1} start_f + run_f = 0 \wedge \bigwedge_{k=i+1}^{n} run_k + preempt_k \leq 1$
   $\rightarrow run_i \wedge wait_k$ [1]

[1] Please note, $wait_k$ is enabled if an according task was running run_k

3. $start_i \wedge \bigwedge_{f=1}^{i-1} start_f + run_f > 0 \vee \bigwedge_{k=i+1}^{n} run_k + preempt_k = 2$
   $\rightarrow wait_i$

4. $wait_i \wedge \bigwedge_{f=1}^{i-1} wait_f + run_f = 0 \wedge \bigwedge_{k=i+1}^{n} run_k + preempt_k \leq 1$
   $\rightarrow run_i \wedge wait_k$ [1]

5. $(clk)\ modulo\ (freq_i - 2) = 0 \rightarrow start_i$

The first rule ensures that only one task is running in a cycle per core. With the second rule, we assure that from a set of tasks to be started only that task with the highest priority is enabled, unless there is no task running with a higher priority or a non-preemptive one. All other tasks also having the $start$ flag set are set to $wait$ according to the third rule. The fourth rule ensures only that waiting task with the highest priority becomes active unless no task with higher priority or a non-preemptive one is running. The last rule guarantees that each task is correctly started according to its frequency. Note, since run is only valid one cycle after $start$ and $start$ also needs one cycle to become active, $start$ has to be set two cycles in advance.

In our model we handle an interrupt service routine (ISR) as a task with highest priority. The only difference compared to a task is that an ISR can occur in every cycle. With this task model we are also able to handle POSIX threads. Each thread can be treated as a task as well, except it only gets started once after its creation, but can still be interrupted.

2.1.2 Efficient Model Unwinding

In this BMC approach the model is not entirely unrolled in each iteration, instead the model is only unwound instruction per instruction. For all instruction types the unwinding process is self-evident, except the branch instructions have to be considered separately. Fig. 4 shows an example of an instruction list containing a branch and a jump instruction. The branch instruction (s_i) has two possible successor instructions. If the condition `cond`

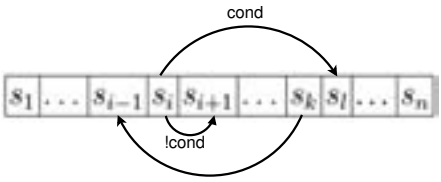


Figure 4: Linked Instruction List containing a Branch Instruction (s_i) and a Jump Instruction (s_k).

is fulfilled the successor instruction is s_l, otherwise s_{i+1}. In SAT-/SMT-Solver a decision made on the root-level (0) is irreversible. All other decisions are hypothetical and can be cancelled. Hence, to unroll a branch instruction based on the condition `cond` two cases have to be taken in account:

1. If `cond` has a pending decision, which was made on decision level (DL) 0:
   $\Rightarrow$ Either if `cond` is true **only** instruction s_l is unrolled,
   $\Rightarrow$ or if `cond` is false **only** instruction s_{i+1} is unrolled.

2. If `cond` has a pending decision, which was made on any decision level greater than 0 or has no pending decision:
   $\Rightarrow$ **Both** instructions s_{i+1} and s_l are unrolled.

If the second case occurs the list of next instructions to unroll increases, but is still a lot smaller than as all instructions were to be unrolled in each iteration.

The described unrolling procedure can also be applied to the task model. Like a branch instruction a task instruction has two possible successor instructions. If it is running the successor instruction is the next instruction in line and if it is waiting it is the instruction itself. In case the variables run and $wait$ of a task have an assignment on DL zero, only the corresponding instruction is going to be unrolled.

2.2 The MEMICS Solver

The solver of MEMICS is an Interval Constraint Solver, for which the overall solve process is shown in Fig. 5.

```
01: bool solve() {
02:   if (!init())
03:     return false;
04:   do {
05:     if (!update())
06:       result = false;
07:     else
08:       if (endOfFile)
09:         return false;
10:       result = search()
11:   } while (result != true);
12:   return true;
13: }
```

Figure 5: The Solve Process of MEMICS.

First, the solver gets initialized in function `init()` (Fig.5 line 02) with the registers of the model and the initial instructions. Then the actual procedure begins, which runs until either a runtime error has occurred or no further instruction is reachable. The unrolling process of the model is represented by the function `update()` (Fig.5 line 06). In this procedure the successor instruction(s), which are reachable from the current state(s) of the model, are transformed into a set of clauses in conjunctive normal form (CNF). The variables occurring in these clauses are in SSA form. If this set of clauses is empty no further instructions are reachable, indicating the safety of the model. Otherwise the search

process is invoked. In case the search also did not find any runtime error, the procedure continues with the next unrolling step. If the process has finished without finding any runtime error, the model and the corresponding source code is free of runtime errors. Please note, this result is only valid for those runtime errors our solver can currently detect. All discovered errors are architecture specific. There might exist architectures on which errors, that we find, do not hold. We also have to state that there is currently no support for passing specific bounds for the unrolling of internal loops, but the user can set an overall model unrolling bound.

```
01: bool search() {
02:   while (true) {
03:     conflict = icp();
04:     if (conflict) {
05:       if (DL == 0)
06:         return false;
07:       level = analyze(conflict, learnt);
08:       backtrackAndAddLearnt(level, learnt);
10:     } else {
11:       if (!decide())
12:         return true;
13:     }
14:   }
15: }
```

Figure 6: The Search Process of MEMICS.

Fig. 6 illustrates the overall search procedure. Initially, interval constraint propagation is used in function `icp()` (Fig.6 line 03) to apply the new interval deductions to the current system, which occurred while adding the new clauses in `update()` (Fig. 5 line 05). If this propagation leads to a conflict on decision level (DL) zero, a conflict on the root level has been found. This indicates that the formula of the current iteration is unsatisfiable. If the conflict does not occur on the root level, the reasons for this conflict are computed in `analyze()` (Fig.6 line 07). The result of this process is a clause eliminating this conflict for the succeeding runs, which is added to the system. On the other hand if no conflict occurred, there also exist two possible cases. If there exists no assignment for all variables leading to a run time error, which means that the model is "safe" (in the current iteration) and the solve process continues. If not, the next variable is picked from the variable queue, a new interval decision is computed for it and the process continues in line 3.

2.2.1 Memory Access Handling

In order to handle memory access the instruction set of the solver contains load and store instructions, as well as malloc and free instructions.

The load instruction (32-bit value) is defined as:

lw <target-register, base-address + offset>

The target-register is a common 32-bit variable and the base-address + offset the address inside the memory vector of the model. The base-address is either the current value of the stackpointer register sp, or it can also be a value stored in a common register. If the resulting address from base-address + offset is outside the allocated range of the base-address an index out of bound error is thrown. When interpreting the instruction, the solver reads the 4 bytes starting at the base-address and assigns the corresponding value to the target-register. The load instructions lb (8-bit value) and lh (16-bit value) operate likewise.

The syntax of the store instruction (32-bit value) is defined as:

$$sw \text{ <source-register, base-address + offset>}$$

This instruction as well as sb (8-bit value) and sh (16-bit value) are working analogously to the load instructions.

2.2.2 Memory Allocation

In order to support dynamic memory management, we have equipped the solver with the table $MT = \{m_1, \ldots, m_n\}$, where an entry m_i is defined as the tuple $m_i =$<address, size>. The instruction set has also been extended with the instructions malloc and free. The new instructions are defined as:

$$malloc < \text{target-register, size} >$$
$$free < \text{base-address} >$$

Whenever a malloc instruction occurs, the solver consults the table, returns the first available address by considering the required size and marks that area occupied by inserting an entry into the table. In case of a free instruction, the solver searches for the base-address inside the table. This search process can lead to three different results:

1. The base-address is found and the memory addresses according to the size entry in the table get cleared.
2. In case the base-address is found but at least one of the memory addresses is already freed, which results in a double free error.
3. If the base-address is not found, it results in an invalid free error.

2.3 Race Condition

A race condition [NM92] is a type of defect, where access to shared data among different tasks/threads is leading to undefined behaviour of the software itself. One kind of race condition is a lost update, which is described in the following example.

Lost update example Assumed there are two tasks, task A and task B, which are concurrently executed on different cores. Task A is reading a global variable $ext1$ twice. But in between the to read operations task B has written an new value to $ext1$. Unless that behaviour is claimed, task B continues with a corrupt value. Using the instruction set of the MEMICS model a load and store transition can have the logic encoding:

$$s_{i'} \wedge run_task_i \rightarrow lw(dest, ext1) \wedge lw_task_i_ext1_k = clk_x \wedge s_{i''}$$
$$s_{j'} \wedge run_task_j \rightarrow sw(value, ext1) \wedge sw_task_j_ext1_l = clk_y \wedge s_{j''}.$$

Using this encoding, the following rule has to hold for $o > m$ in order to find a lost update error as described above:

$$lw_task_i_ext1_o - lw_task_i_ext1_m < freq_task_i \wedge$$
$$lw_task_i_ext1_o > sw_task_j_ext1_p \wedge sw_task_j_ext1_p > lw_task_i_ext1_m$$

3 Results

The implementation of the MEMICS tool was tested on a test set containing different types of runtime errors based on the error classes from the Common Weakness Enumeration (CWE) [cwe] database. In Table 1 the results of a comparison between our tool and CBMC and LLBMC are shown.

Class	Benchmark	CWE-ID	MEMICS	CBMC	LLBMC
Arithmetic	DivByZeroFloat	369	✓	✓	
	DivByZeroInt	369	✓	✓	✓
	IntOver	190	✓	✓	✓
Memory	DoubleFree	415	✓	✓	✓
	InvalidFree	590	✓	✓	✓
	NullDereference	476	✓	✓	✓
	PointertToStack	465	✓	-	✓
	SizeOfOnPointers	467	✓	-	✓
	UseAfterFree	416	✓	-	✓
Pointer Arithmetic	Scaling	468	✓	-	✓
	Subtraction	469	✓	-	✓
Race Condition	LostUpdate	567[2]	✓		
	MissingSynchronisation	820	✓		
Synchronization	DeadLock	833	✓		
	DoubleLock	667	✓		

Table 1: Results of MEMICS compared to CBMC and LLBMC

The first column of the table shows the overall class the test cases reside in and column two shows their subclasses. In the third column the official CWE-ID of each subclass is given.

[2]We did not find a straight forward ID for a lost update, but the example in this entry describes one

With that id, one can look up the detailed description of the runtime error on the CWE website. Column three shows the results of our MEMICS tool, compared to CBMC in column four and LLBMC in column five, where ✓represents a correct verification result, - a false one and a whitespace signals that the tool does not support the class of the test cases.

The results provided in Table 1 show that our MEMICS tool is already able to handle a wide range of test cases and in contrast to BMC tools like CBMC and LLBMC is able to identify concurrency issues like race conditions and synchronization problems. In order to give an example on how our tool is dealing with a task based system, Fig. 7 shows a portion of C code implementing a global variable `signal`, a global event `trigger` and two tasks, a $1ms$ and a $100ms$ one. The $1ms$ task is first reading from the global variable

```
01: int signal;                    10: void task_100ms()
02: event trigger;                 11: {
03:                                12:   int x = signal;
04: void task_1ms()                13:   wait_on(trigger);
05: {                              14:
06:   int x = 2*signal;            15:   if (signal < 0)
07:   signal = 1;                  16:     x = -x;
08:   ...                          17:   ...
09: }                              18: }
```

Figure 7: Simple Task Based System: task.c

`signal` (line: 06) and then assigns 1 to it (line: 07). The $100ms$ task assigns the value from `signal` to its local variable `x` (line: 12) and is, in the following, waiting for the event `trigger` (line: 13). Once the event occurred, the $100ms$ task checks if the value contained in `signal` is negative (line: 15). If it is negative the previous local variable is assigned with its absolute value (line: 16). This scenario describes a lost update error, as described in the example in Section 2.3. The configuration for the system of the task

```
BASE {                             TASK {
  NUMOFCORES = 2,                    NAME = task_100ms,
  FREQUENCY = 100,                   FREQUENCY = 100,
  MEMSIZE = 256                      PRIORITY = 1,
};                                   CORE = 1,
TASK {                             };
  NAME = task_1ms,                 EVENT {
  FREQUENCY = 1,                     NAME = trigger,
  PRIORITY = 1,                      FREQUENCY = 0.1
  CORE = 0,                        ;
};
```

Figure 8: System Configuration File to the Task Example: example.conf

example (Fig. 7) is shown in Fig. 8. The system features a processor with two cores

running at $100MHz$ and $256MB$ of main memory, assigns the $1ms$ task to the first and the $100ms$ task to the second core, and defines the event `trigger` to be raised every $100\mu s$.

The resulting output of our MEMICS tool applied to the task example from Fig. 7 with the configuration from Fig. 8 is shown in Fig. 9. It shows that our tool finds a lost update error as described above. On top of that our tool is able to print the entire trace leading to the error by adding the option `-print-counterexample`. Currently this output is only provided in a MIPS like assembly style and is therefore not shown here, but a mapping back to the C/C++ code is work in progress.

```
user@memics:~/$ memics -task-conf=example.conf task.c

Model is CORRUPTED
Run time error: lost update
  occurred in file: task.c @ line: 15
        if (signal < 0)
  in conflict with statement @ line: 9
        signal = 1;
  triggered from line: 12
        int x = signal;
```

Figure 9: MEMICS called on the Task Example Code.

4 Conclusions and Future Work

In this paper we have introduced the new MEMICS tool for software verification. Its core is a new Interval Constraint Solver, that is directly operating on a machine code based model and is capable of handling memory access internally. In addition, the model, our tool is working on, features an efficient model unrolling strategy. The results from Section 3 have shown that our tool can already verify a large number of error classes. In comparison to tools like CBMC and LLBMC, our tool is able to handle software implementing a task-based operating system like AUTOSAR or OSEK. On top of that it is already capable of handling concurrent software. The last point is essential, as the growing application of multicore processors especially in the field of embedded devices, leads to the need of a method for the verification of concurrent software.

The main goal we want to achieve in the near future is to test our tool on industry software in order to check its scalability. Another huge aspect is to find a suitable partitioning strategy for our model, to run the verification process itself in parallel. This task is of large relevance as the complexity of concurrent software is, compared to sequential software, growing very fast, especially with increasing size.

References

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. The ASTRE analyzer. In *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[Con] Autosar Consortium. *AUTOSAR - Specification of Operating System*. http://autosar.org.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[cwe] *Common Weakness Enumeration*. http://cwe.mitre.org.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[dMB09] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: An Appetizer. In Marcel V. Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902, chapter 3, pages 23–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Fan10] Dominic Fandrey. Clang/LLVM Maturity Report. June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[IYG+04] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *in Symposium on Leveraging Formal Methods in Applications*, 2004.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat] Chris Lattner. *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[ose] *OSEK*. http://portal.osek-vdx.org.

[pol] *Polyspace*. http://www.mathworks.com/products/polyspace.

[SFM10] Carsten Sinz, Stephan Falke, and Florian Merz. A Precise Memory Model for Low-Level Bounded Model Checking. In *Proceedings of the 5th International Workshop on Systems Software Verification (SSV '10)*, Vancouver, Canada, 2010.

[Swe06] Dominic Sweetman. *See MIPS Run, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Tra10] Johannes Traub. Program Analysis and Probabilistic SAT-Solving. Master's thesis, Universität Stuttgart, 2010.

Freedom from Interference for AUTOSAR-based ECUs: a partitioned AUTOSAR stack

David Haworth, Tobias Jordan, Alexander Mattausch, Alexander Much

Elektrobit Automotive GmbH, Am Wolfsmantel 46, 91058 Erlangen, GERMANY

Abstract: AUTOSAR[1] is a standard for the development of software for embedded devices, primarily created for the automotive domain. It specifies a software architecture with more than 80 software modules that provide services to one or more software components.

With the trend towards integrating safety-relevant systems into embedded devices, conformance with standards such as ISO 26262 [ISO11] or ISO/IEC 61508 [IEC10] becomes increasingly important.

This article presents an approach to providing freedom from interference between software components by using the MPU[2] available on many modern microcontrollers. Each software component gets its own dedicated memory area, a so-called memory partition. This concept is well known in other industries like the aerospace industry, where the IMA[3] architecture is now well established.

The memory partitioning mechanism is implemented by a microkernel, which integrates seamlessly into the architecture specified by AUTOSAR. The development has been performed as SEooC[4] as described in ISO 26262, which is a new development approach. We describe the procedure for developing an SEooC.

[1] AUTOSAR: AUTomotive Open System ARchitecture, see [ASR12].
[2] MPU: Memory Protection Unit.
[3] IMA: Integrated Modular Avionics, see [RTCA11].
[4] SEooC: Safety Element out of Context, see [ISO11].

Introduction

The complexity of the functions that the software of an embedded device performs in a car increases significantly with every new car generation. AUTOSAR is an approach to manage the complexity by providing a standardized software architecture.

A further trend is the integration of several different functions on a single device, which has been simplified significantly by the AUTOSAR concept, as it defines a standard configuration and runtime environment for all suppliers of software components. It defines XML-based data exchange formats that the various suppliers can use to exchange their software components and data.

In this scenario, several software components are provided by different parties involved in the development. During integration, it is necessary to ensure that the different components do not interfere with each other. Interference between software components is not only a problem during development; it can also have legal consequences. This problem becomes especially important when software components with different criticality need to be integrated, for example with different safety integrity levels (SIL[5]/ASIL[6]). These highly integrated embedded devices are often referred to as *domain controller* – a central device that is responsible for all computation of a certain car domain.

We present an approach that integrates seamlessly into the AUTOSAR software architecture and provides freedom from interference in the spatial domain based on the highest (A)SIL levels.

[5] SIL: Safety Integrity Level, see [IEC10].
[6] ASIL: Automotive Safety Integrity Level, see [ISO11].

The AUTOSAR architecture

The AUTOSAR environment is a layered software architecture with about 80 separate modules. The general architecture is shown in Figure 1. The lowest layer contains the hardware-dependent modules, the microcontroller abstraction layer (MCAL) together with the operating system (OS). The basic software (BSW) modules are placed on top of the MCAL and are hardware-independent. The interface to the application layer is provided by the runtime environment (RTE).

The application layer is comprised of several independent entities, the *software components* (SWC). The SWCs themselves contain the tasks and ISRs that provide the functionality of the ECU. The communication between SWCs and with the underlying BSW is handled by the RTE, providing a complete abstraction from the drivers and the hardware and allowing completely hardware-independent application development. The protection measures for safety-relevant functionality, for which a certain safety integrity level is required, are in general also contained within the application.

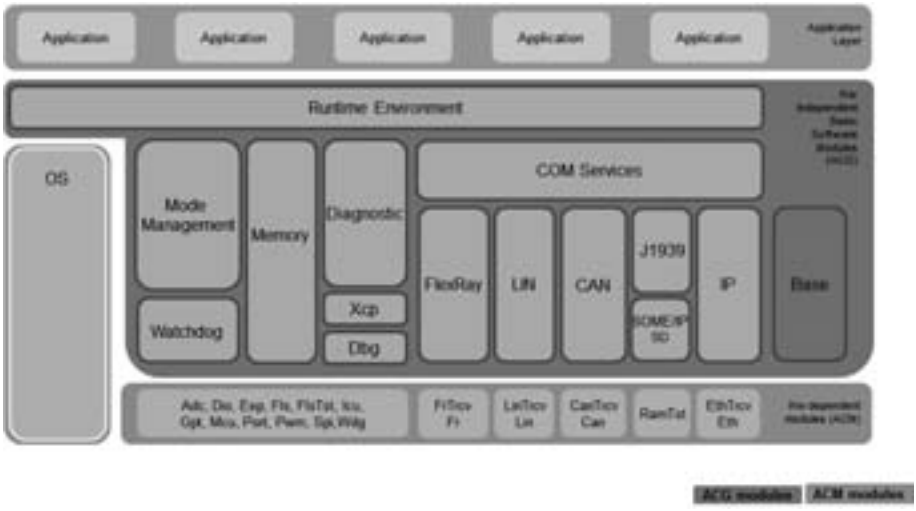


Figure 1: The AUTOSAR software architecture (Release 4.0)

However, although the SWCs are decoupled and independent of each other, the standard AUTOSAR architecture assumes that the entire basic software, including OS and RTE, runs within a single privileged context. Thus, although the safety mechanism of the device may reside within a single SWC, this SWC depends on the correct execution of the entire AUTOSAR stack. Freedom from interference is therefore not guaranteed, and as a consequence not only the entire OS but also the entire AUTOSAR stack inherits the highest integrity level.

A solution to this dilemma would be to establish freedom from interference between the SWCs and the BSW using the OS. The following sections describe such an approach.

A partitioned AUTOSAR stack

A common approach to separating software components from each other is partitioning. The software components run in isolated areas and any interference is inhibited by a memory protection mechanism. In safety-critical environments, partitioning allows the integration of software components of different (A)SIL levels on a single system.

The partitioning mechanism is best placed in the operating system, which will be described in further detail in Section 0. However, from the application's point of view, the operating system is the entire AUTOSAR stack, comprising all modules including and below the AUTOSAR runtime environment (RTE). The AUTOSAR stack is very complex software with several thousand configuration parameters and functionality that exceeds by far the typical necessities for the safety mechanism that is implemented in the system. It is thus neither technically nor commercially feasible to develop the entire AUTOSAR basic software (BSW) according to the highest safety integrity level which is required for the entire system.

Instead, we extend the partitioning approach to the AUTOSAR stack itself. The stack is split into two parts:

- The operating system, which is assumed to provide the context-specific memory protection and to have been developed according to the highest integrity level
- The AUTOSAR BSW, which is encapsulated in a separate memory partition with limited rights and cannot interfere with the remaining parts of the system.

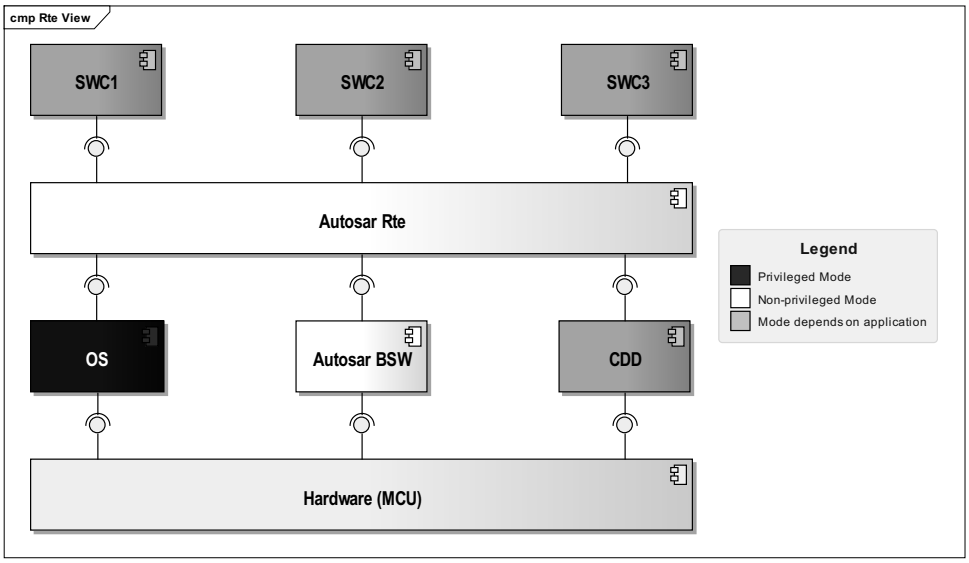


Figure 2: Architectural overview of AUTOSAR modules and software components

This approach allows the assignment of only a few safety-related requirements onto the operating system, namely the provision of safe task and interrupt scheduling and a safe memory protection mechanism, while for all other components usually no safety requirements apply. Therefore, these modules can be developed with usual QM approaches.

Timing and communication mechanisms are not part of the partitioning concept. If the safety concept requires time and control flow monitoring or protection of the communication, they must be part of the application. This can be achieved with standard modules provided by the AUTOSAR stack: the watchdog manager interface provides mechanisms for time and control flow monitoring in conjunction with an external watchdog, while the end-to-end protection library protects the communication channels.

The architecture of such a partitioned AUTOSAR stack is shown in Figure 2. In contrast to Figure 1, the entire basic software including the MCAL is contained in the component *Autosar BSW*. In this scenario, only the operating system runs in the privileged mode, while all other components run in the non-privileged user-mode or have at least the possibility to do so. In addition to three software components, Figure 2 shows also a complex device driver (CDD). All these components can run in the non-privileged mode, although this is the decision taken in the software architecture.

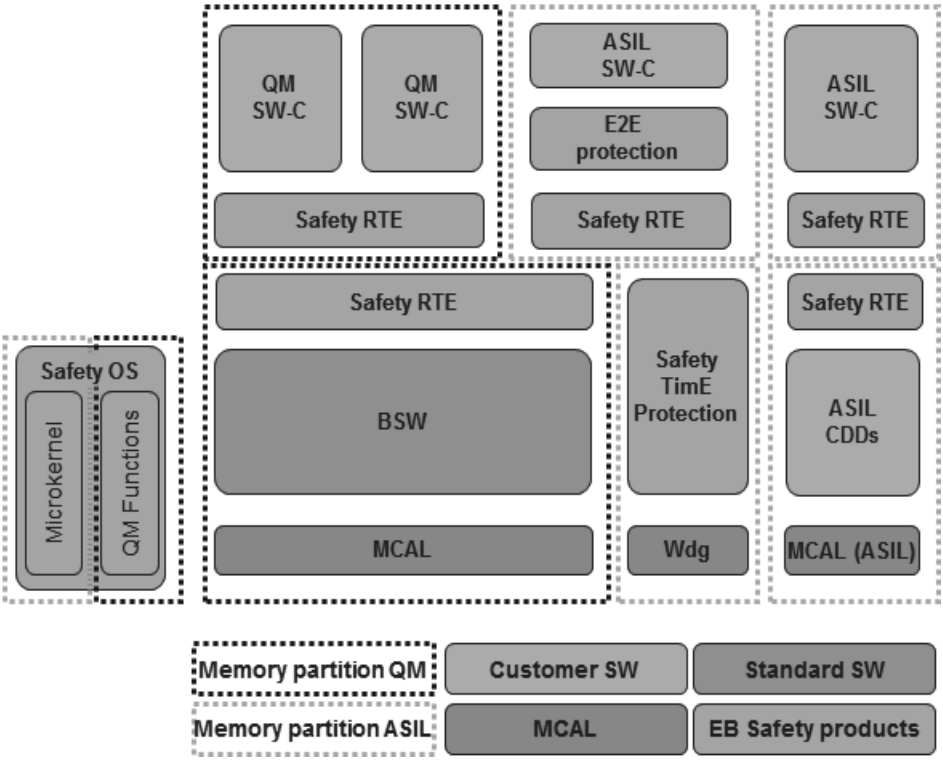


Figure 3: Safety architecture for an AUTOSAR ECU

The approach of putting the AUTOSAR BSW into a separate partition running with reduced rights has the advantage of drastically reducing the code that has to be developed, analyzed and verified according to the highest safety integrity level. Freedom from interference on the data level between the software components and the AUTOSAR BSW is provided by the memory protection which is implemented in the operating system.

Two prerequisites are necessary for this setup:

- Since the MCAL is placed in the AUTOSAR BSW component, which runs in the non-privileged mode, the CPU must grant access to the peripherals in this mode. However, modern CPUs for safety applications have considered this and allow such an operation. Otherwise, the MCAL would have to be adapted to request the elevated rights from the operating system before accessing the peripherals.
- Timing protection and control flow monitoring are still part of the application. The interface to an intelligent watchdog driver can be used by the application.

An exemplary safety architecture detailing a partitioned AUTOSAR ECU including a partitioned RTE, time and execution protection and end-to-end communication protection is shown in Figure 3.

The microkernel concept

A full implementation of the AUTOSAR Os module would provide memory protection for tasks and interrupt service routines (ISR) to the standards required for safety-critical software. However, the OS module is a very large piece of software that would be costly and time-consuming to implement to the high standards required for safety-critical software.

An alternative solution of using a standard OS with additional protection measures has been proposed. The protection features required to ensure freedom-from-interference between tasks and ISRs are listed in Table 1, along with the external protection measures that would be necessary when using a standard OS.

However, it has already been observed [Ha11] that in the presence of an operating system it is not sufficient merely to program the MPU to permit access to the critical data during the computation that uses it. This is because any software that executes while the access is permitted, including interrupt service routines and tasks of higher priority, could modify the critical data without detection.

The design of the standard AUTOSAR Os specifies re-entrant functions and allows the execution of ISR routines while the MPU is programmed. The Safety Os however doesn't allow any interruptions or parallel execution while access to the MPU is allowed, leading to the differences listed in Table 1.

Table 1: Safety properties of the standard AUTOSAR Os and the Safety OS

	AUTOSAR Os	Safety OS
Setting the MPU	Yes	Yes
Correctly setting the MPU	No, requires additional measures, e.g. regular value checks	Yes
Safe handling of critical registers (PC, SP, μP mode)	No, requires additional measures, e.g. control flow monitoring	Yes
Safe handling of critical memory regions (stacks, context safe areas, Os states)	No, requires additional measures, e.g. control flow monitoring	Yes
Safe task switch, pre-emption and resume	No, requires additional measures, e.g. control flow monitoring	Yes
Safe interrupt/exception handling	No	Yes
Correct task/interrupt priority based scheduling	No	Yes
Safe deactivation of faulty subsystems	Yes, reset into error mode	Yes, reset into error mode
Safe resource management	No	Yes
Safe event management	No	Yes

Furthermore, there are failure modes in an operating system that can have very subtle effects on the integrity of the tasks that it controls. These effects would be extremely difficult or impossible to detect using the proposed measures. Even the provision of a high-integrity memory protection driver, loosely coupled to a standard operating system, would not provide sufficient protection. This is because during the critical computation the data is not confined to memory but can reside in the processor's registers, which are fully under the control of the operating system.

When other factors such as the integrity of the stack and the critical section control are taken into account, it rapidly becomes clear that the operating system itself must provide the protection.

Fortunately, the OS has many features that could easily be separated from the core of the operating system using the same mechanism that provides the freedom from interference between tasks and ISRs. The minimum core of the OS that must be developed to the high standards for safety critical software is called a microkernel.

In principle, the microkernel should be the only software that runs in the privileged mode of the processor, and is therefore the only software that can modify the contents of the MPU. The main functionality of the ECU, including communication and I/O, along with OS features that provide time-triggered activation, run at a lower privilege level ("user mode") in which access to the MPU is forbidden by the processor. This software need not be developed to the same standards as the microkernel.[7]

The defining feature of the microkernel is that it uses a hardware feature called a *system call*, which is typically a vectored exception, to control transitions between tasks and ISRs and the microkernel. In addition to the system call, all other transitions into privileged mode, such as interrupts and processor exceptions, are handled by the microkernel in a similar way. Using this mechanism, it is not possible to switch from user-mode into privileged mode without simultaneously transferring control to the microkernel.

The microkernel manages all the executable objects in the system. The set of executable objects contains:

- Tasks

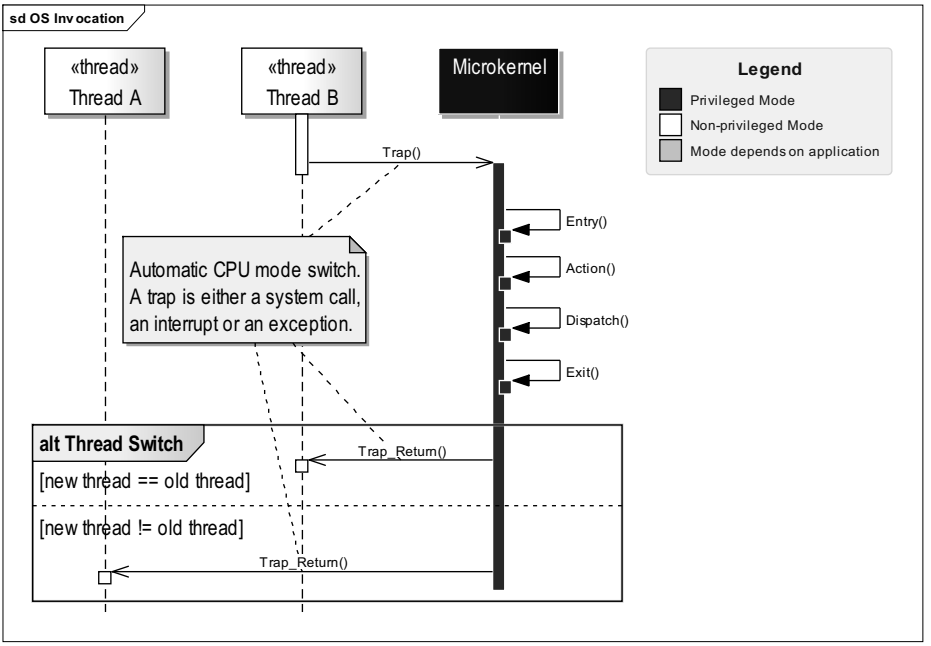


Figure 4: Control flow upon microkernel activation

- ISRs
- Hook functions
- Time-triggering services provided by the standard OS
- Idle loops
- Startup: the `main()` function.

These objects are all managed using a mechanism called a *thread*.

By means of the system call mechanism, it can be demonstrated that the behavior of a user-mode thread with respect to the data of another thread is not merely "acceptably safe", or that interference is "unlikely" or "improbable"; rather, it can be demonstrated that interference cannot occur.

The control flow through the microkernel is depicted in

Figure 4. The kernel function that is called depends on the type of interrupt or exception, and may result in a different thread becoming more eligible to use the CPU than the current thread. The change of thread is performed by the dispatcher, and includes reprogramming the MPU to the correct settings for the incoming thread.

This simplified control flow means that the microkernel can be made non-reentrant, thus further simplifying the design of the microkernel and thus the assurance of correctness.

This also means that the microkernel is only concerned with the execution of thread switches and programming the MCU. All other functions of a standard AUTOSAR Os are outside of the scope of the microkernel and considered QM functions.

Partitioning with an AUTOSAR operating system

From an AUTOSAR operating system point of view, SWCs are just applications using the OS interface, since the RTE mainly encapsulates the OS and BSW APIs within its own layer. For the OS, the SWCs are mapped to so-called *OS Applications*, as shown in Figure 5.

OS Applications are a container for tasks, ISRs and any other OS controlled objects that belong to a certain application. They are also the boundary for the memory partitioning: all objects belonging to the same OS Application can share common data.

Consequently, Figure 5 shows seven different memory partitions: three SWCs, a CDD, the AUTOSAR BSW, the QM part of the operating system and the microkernel itself (although running in the privileged mode, the kernel grants access only to its own data).

The configuration of the OS Application also decides on the CPU mode in which its tasks and ISRs are executed. Depending on an application-specific configuration

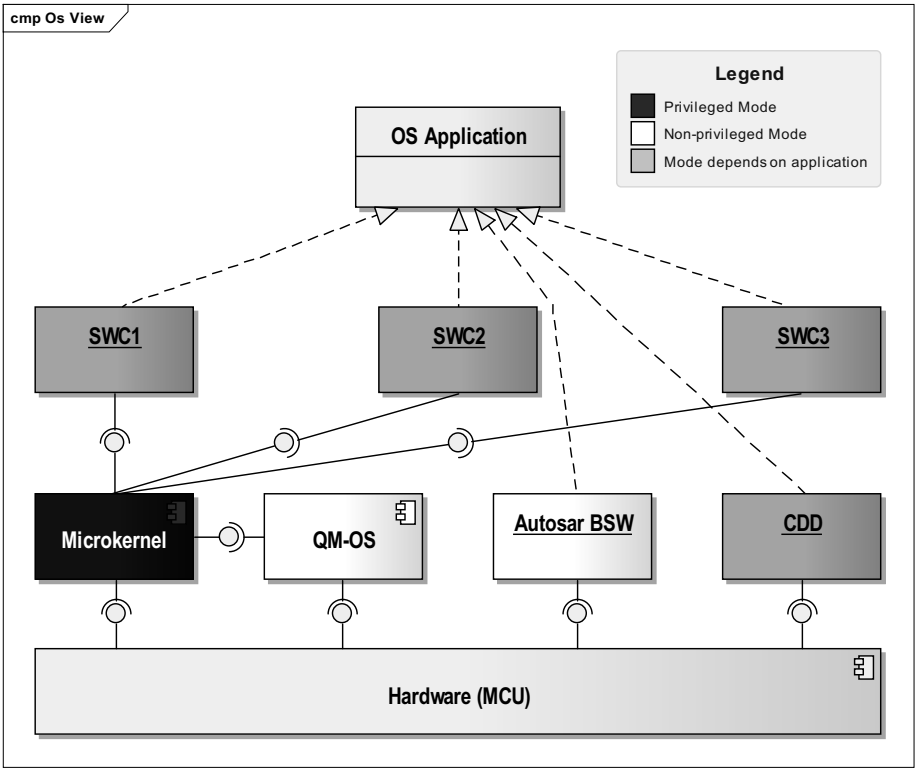


Figure 5: AUTOSAR stack configuration as seen from the OS module

parameter, all application objects either use the CPU's privileged mode or the non-privileged mode.

Running components in the non-privileged mode gives the highest protection against any kind of error: the MPU providing the foundations of the partitioning mechanism is protected against any change, only the microkernel is allowed to reconfigure the memory protection. The microkernel itself is only entered via defined access points – namely the interrupt and exception interface.

An important goal for the design of the Safety Os was to integrate it seamlessly and directly into the AUTOSAR architecture. Given AUTOSAR as a fixed standard this means that all requirements specified by AUTOSAR still need to be fulfilled. This could be achieved and starting with a standard AUTOSAR architecture the Safety Os can be added to an AUTOSAR Os without affecting any AUTOSAR requirement apart from the operating system itself.

The partitioning via software components (SWC) and their OS Application counterpart is a well-established technique in safety-related development. Similar approaches are known in aviation as the IMA architecture [RTCA05].

Implications for the development process

The development of a safety-related operating system introduces challenges into the development process. From the start, the development process was based strongly on Automotive SPICE in the HIS [HIS12] scope. The challenge was to extend the existing processes to cope with the additional requirements arising from safety-related development according to ISO 26262 and IEC 61508.

Since the projects are regularly assessed in project audits, it was decided to extend these audits by introducing essential safety aspects.

Two main lessons have been learnt during the development:

- The HIS scope of SPICE is not sufficient to support safety development
- Internal SPICE assessments have been updated with additional checklists in order to check method definitions and their usage. Such assessments can then be used as functional safety audits according to ISO 26262.

In order to support safety development the process scope has been extended from the HIS scope to cover more processes. In the following, we describe the development of the OS as a safety element out of context and focus then on process extensions which have been implemented to fulfil safety requirements.

Safety Element out of Context (SEooC)

The increasing complexity of modern software projects encourages the use of reusable software components (RSC) with clearly specified functionality. In avionics, the concept is well-established and part of DO-178C [RTCA11]. In ISO 26262, it is referred to as a *Safety Element out of Context* (SEooC). The key element of a SEooC is the set of assumptions that are applied during the development. The approach is shown in Figure 6.

The most important aspect is a clear description of the context in which the SEooC is used and the assumptions that are made. These assumptions define the scope of the SEooC and have impact on its requirements and its design.

Two simple rules have proved to be useful during the development of the OS component:

- The fewer assumptions exist, the clearer is the focus of the component.
- The more precise the assumptions are, the simpler is the implementation.

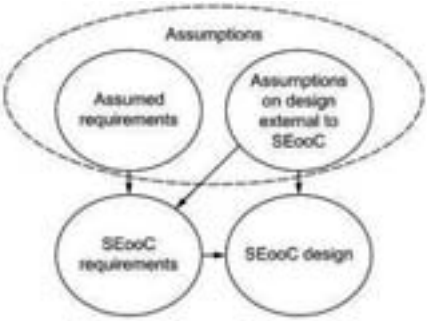


Figure 6: Relationship between assumptions and the development of a SEooC

These rules lead to a small but flexible product which can be used in many different environments. A small set of assumptions also simplifies the usage of the component in the final device. Since the scope of the SEooC is much clearer than with a large assumption set, the safety argument for the entire device becomes much more concise. In addition, this concept allows the re-use of existing safety mechanisms including any certifications that may already be available.

Extended Tracing

In SPICE conformant development processes, the tracing between requirements and their associated test cases is a well-established technique. In a safety-related development, further work products are created. A wide-spread technique to ensure consistency between the content of the various work products is to extend the dependencies between work products to tracing between the content of work products.

For example, to ensure that the correct functionality has been implemented, bi-lateral tracing from requirements through the software architecture and the software unit designs to the source code is already part of Automotive SPICE.

In safety-relevant projects, the number of documents that is required to fully specify and document the implementation as well as its verification can increase significantly: for example, a safety-case providing the safety argument, a hardware-software interface specification and a safety manual that details the correct and safe use of the software product are the most prominent work products.

The foundation of the content of these documents is the software safety requirements specification and each of these documents can contain the fulfilment of a requirement:

- Safety case: by an argument, e.g. a result of a safety analysis
- Hardware-software specification: e.g. does the hardware behave as documented?

- Safety manual: e.g. are all assumptions being taken into account by the software configuration?

As these documents can become an end-point for software safety requirements, from a tracing point of view they need to be treated just like a software unit design or the implementation itself. The tracing does not have to be bi-lateral as for example a safety manual also contains information that is not directly motivated by a software safety requirement.

The tracing model of Automotive SPICE needs to be extended to include such safety work products in order to construct an argument for the completeness of the fulfilment of software safety requirements. See

Figure 7 for an example of a tracing model.

The extension of the tracing model to include safety work products is especially important in SEooC development wherever the final context in which the software is expected to be used is assumed and documented.

Summary and Outlook

In this paper we presented a microkernel based approach for a partitioned AUTOSAR system. In this concept, only the microkernel runs in the CPU's privileged mode, providing freedom from interference in the spatial domain between the AUTOSAR basic software, the integrated software components and complex device drivers.

In order to reduce the size and complexity of the code running in the privileged mode

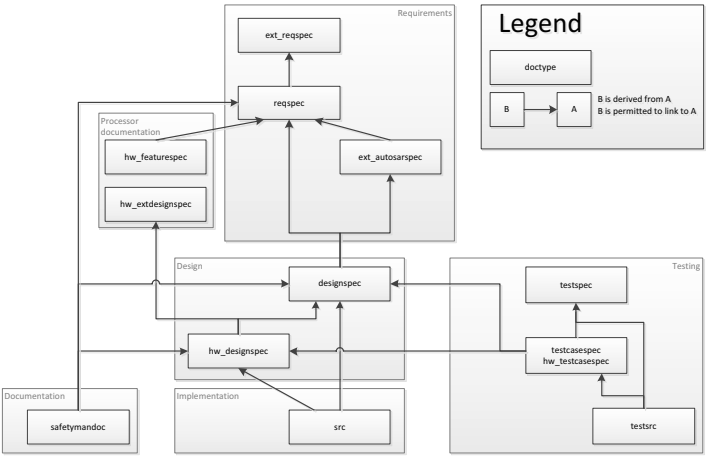


Figure 7: Tracing model of the Safety OS

even further, the operating system has been split into a safety-relevant part providing the basic scheduling features and the memory protection, and a part containing the non-safety relevant functionality which runs in its own non-privileged partition. This reduces to a minimum the size and complexity of the software that needs to be developed according to the highest safety integrity level.

In addition, the development processes of the standard product development have been extended to support the creation and the tracing of the additional work products as required by ISO 26262. They are executed for the development of the OS as a Safety Element out of Context (SEooC), allowing for the reuse of the component in various different environments and systems.The development of the Safety Os including all safety related tasks is already finished and it is now going through a certification process with an external party to have a certification according to ISO 26262 (up to ASIL-D) and to IEC 61508 (up to SIL-3).

With the emergence of more and more powerful multi-core CPUs, partitioning will become even more relevant in the future. The additional computational power will be used to integrate software from various different suppliers often with different quality levels, requiring strict supervision and encapsulation of the supplied components. In addition, domain controllers combining several different functions of an entire car domain are currently being introduced. These devices integrate a multitude of software components, where partitioning is a key concept to ensure the non-interference between the modules. With the development of highly integrated many-core devices this trend will accelerate further.

Bibliography

[ASR12] AUTOSAR, http://www.autosar.org.

[ASPICE10] Automotive SPICE® Process Assessment Model v2.5, May 2010.

[ISO11] ISO 26262, Road vehicles - Functional safety, ISO copyright office, 2011-11-15.

[IEC10] ISO/IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related Systems, IEC central office, 2010.

[MIRA04] MIRA Ltd., MISRA-C:2004 - Guidelines for the use of the C language in critical systems, ISBN 0952415623, October 2004.

[RTCA11] DO-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc., 2011-12-13.

[RTCA05] DO-297, Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, RTCA Inc., 2005-08-11.

[Ha11] Haworth, David: An Autosar-compatible microkernel. In Herausforderungen durch Echtzeitbetrieb. Berlin: Springer-Verlag, November 2011

[HIS12] Hersteller-Initiative Software, http://www.automotive-his.de/

Why current Memory Management Units are not suited for Automotive ECUs

Jörn Schneider
Trier University of Applied Sciences
Schneidershof, 54293 Trier
j.schneider@fh-trier.de

Abstract: A major trend in automotive industry is to enrich driver and passenger experience with an increasing amount of consumer electronics and car-2-x functionality. A close interaction between this added functionality and the classical automotive domains allows for innovations that are valuable to the end customer and cannot be outplayed easily by devices with a pure consumer electronic origin. Innovations of this class require a tight coupling, for instance by executing programs from both worlds on the same microprocessor. The latter introduces many challenges, especially regarding reliability, security and safety of such systems. A unified memory management fulfilling the requirements of the consumer electronics and automotive application could help to address these issues and is a challenge by itself. This paper shows that the prevailing implementation scheme for memory management units (MMUs) is not suited for the needs of such systems and points out a solution direction.

1 Introduction

A common foundation of reliable, safe and secure systems are hardware based mechanisms to protect memory regions. In standard IT and consumer electronics devices memory management units provide this service. While in classic automotive domains still only memory protection units are available for this purpose.

Innovations going beyond the state-of-the-art in driver experience require a close interaction between consumer electronics or car-2-x functionality and the classical automotive domains such as engine control, chassis systems, body control, or occupant safety. Only products and services based on a tight integration cannot be outplayed easily by devices with a pure consumer electronic origin. A promising way to achieve this is to execute consumer electronics/car-2-x and classical automotive control programs on the same microprocessor while sharing common data and peripherals.

Executing automotive control and consumer electronics/car-2-x software on the same microprocessor introduces many challenges, especially regarding reliability, security and safety of such systems. A unified memory management fulfilling the requirements of the consumer electronics/car-2-x and automotive domain could help to address these issues and is a challenge by itself.

Developers of standard IT systems or of smartphone software are used to the benefits

delivered by memory management units. In the automotive domain the situation is quite different for several reasons. A primary issue here are costs per Electronic Control Unit (ECU). In this high volume market with small revenue margins the extra costs for increased die size are a serious issue. Nevertheless, if memory management units would have been really required in automotive systems, they would have been there. This was simply not the case in classic automotive domains.

There are two hard reasons a computer scientist can put forward in favor of MMUs:

Memory Protection In a system without memory protection one erroneous piece of code might cause a complete system failure. MMUs allow to protect each process against another.

Management of Secondary Storage (swapping) Transferring code or data between main memory and secondary storage manually is a cumbersome and error-prone activity. Thanks to memory management units, the times where a lot of development effort went in to partitioning of programs into overlays and managing these without operating system support, are past for almost all systems.

None of these two reasons was an issue that required MMUs for classic automotive systems so far. Before the AUTOSAR Operating System specification included memory protection as an optional feature, ECUs where usually developed as closed systems without memory protection. Nowadays memory protection is used in some cases, e. g. due to the integration of applications with different ASILs according to ISO 26262 and this is typically achieved with memory protection units (MPUs).[1] Thereby, memory protection is realized with less die-space-consuming and cost-intensive hardware.

Management of secondary storage is still not necessary, because ECUs use NOR-Flash technology, which allows for execution in place, i. e. the code is executed out of the secondary memory immediately. Thus RAM is required for nonconstant data only.

Systems executing consumer electronics/car-2-x applications and safety-relevant control applications with hard real-time demands on the same hardware, fall into the class of so-called mixed-criticality systems, that exhibit diverging requirements regarding safety and real-time behavior. Within the paper the term hybrid system is used to denote such systems. The state-of-the practice solution is to use different processors for different parts, e. g. an application CPU running a general purpose or soft real-time operating system such as Linux or QNX, and a second CPU as network controller running AUTOSAR OS that handles the communication with other ECUs via CAN or FlexRay bus systems. A different, widely propagated, approach is to execute the involved operating systems on separate virtual machines [HH08, GWLC12]. However, as observed by Gu et al. the overhead of traditional virtualization approaches is usually not acceptable in automotive ECUs [GWLC12].

The virtualization based approach uses a different class of hardware than ECUs. It includes MMUs and many other features foreign to classic automotive systems. Leaner approaches are conceivable that execute consumer electronics/car-2-x applications and control applications on one CPU (with or without multiple processor cores) without full-blown vir-

[1] Memory Protection Units can monitor memory references to ensure that a piece of code accesses only the allowed memory parts and only in the permitted mode (e. g. read-only).

tualization. Such systems would allow to use shared memory as communication media without the usual time-triggered message scheme as it is already established in avionics systems [ari97, CRA+09] and proposed for automotive systems by different vendors. Using shared memory without indirection has the potential to eliminate the high latencies and resource demands of the existing approaches but requires new concepts to maintain real-time guarantees. Some insights on this issue can be found in [Sch12]. Both, the virtualization based solution and the immediate usage of shared memory assume one underlying approach to memory management.

The following Section presents the major requirements on memory management units in future automotive systems from two perspectives. The first viewpoint is that of consumer electronics applications and the second of classical automotive applications. Section 3 gives a brief description of the prevailing MMU approach (i. e. page based MMUs) in standard IT systems. It follows a detailed discussion on dimensioning the page size in order to meet the requirements of future automotive systems in Section 4. A major part of this discussion are quantitative considerations regarding memory efficiency by examining number experiments with exemplary task sets. Finally Section 5 summarizes the made observations and points towards an alternative approach to MMUs that is much closer to the requirements of future automotive systems than the prevailing approach to memory management in standard IT.

2 Requirements on MMUs in automotive systems

This Section describes some central requirements on MMUs from the two major perspectives of future hybrid systems in automobiles: consumer electronics and classical automotive.

2.1 Consumer Electronics Perspective

An MMU in consumer electronics devices fulfills three essential needs:

Logical Address Space The key innovation of MMUs is the separation of the two concerns addressing in a program and location in physical storage of code and data. This achieves the degree of freedom that allows the programmer to abstract from the details of memory management. Additionally, separating these concerns, delivers mechanisms to relocate code and data, and to protect the address space between processes.

The prominent underlying feature of an MMU based system is that each process owns its private address space. This is a logical address space since processes access code and data as if these addresses would reflect the actual physical position in memory although this is not necessarily the case. The mapping of logical addresses to physical addresses is handled by the MMU and the operating system. The application developer considers the level of logical addresses only and abstracts from physical memory layout.

Memory Protection Because every process has its private address space and the MMU divides this address space in different areas (e. g. memory pages of equal size), it is straight forward to assign access rights to areas according to protection needs. Some areas might for instance be marked as read-only or execute-only. A violation of the rights causes the MMU to signal an interrupt to the CPU thereby allowing the operating system to handle the situation. This mechanism is one of the essential foundations for establishing secure systems.

Management of secondary storage Consumer electronics devices typically use technologies for secondary storage that do not offer the possibility to execute code immediately from secondary storage, e. g. NAND flash memory, or disk drives. All programs have to be transferred to main memory (e. g. DRAM or SRAM) to be executed. Usually, there is not enough main memory to hold all programs to be executed completely all the time. Only pieces of different programs might be present in main memory at a given time.

Before MMUs where available a software developer had to manually divide each program into chunks (overlays) that need not be in memory at the same time, and to write code to manage loading of overlays. In current consumer electronics devices the MMU handles these issues together with the operating system.

Note that, the process of fetching needed memory contents from secondary storage into main memory is called swapping in standard IT systems. This includes writing back of changed memory contents to secondary storage if needed. However, this is rarely supported in embedded devices, because disk drives are usually not available. Therefore, only memory content that is unaltered, e. g. code, is replaced by newly fetched memory contents, if no free space is left in main memory.

2.2 Automotive Perspective

The state-of-the-art in electronic control units of the classical automotive domains is as follows:

No Logical Address Space Only physical addresses are used, there are no hardware supported memory abstractions in place.

Memory Protection by MPUs Memory can be protected by memory protection units (MPUs). Each access to a physical memory address is monitored by the MPU that verifies whether the access is in line with the access rights of the executed task. An interrupt is raised by the MPU if this is not the case and the operating system handles the situation as specified by the application developer.

Execution in Place The secondary storage of ECUs uses NOR-Flash technology. This allows to execute code immediately in secondary storage without transferring it to the main memory. Therefore, no hardware support to transfer code into RAM is necessary.

In future systems that integrate applications from both worlds, it is very likely that the described situation changes completely. A logical address space with hardware supported memory abstraction could be very beneficial for automotive systems also, e. g. because the

integration efforts could be reduced and software updates in the field would be easier. Even when MMUs are introduced due to other reasons, it is very likely that they will eventually be used for this purpose.

Memory protection becomes even more important in hybrid systems and is in principle achievable with an MMU, although there are some important differences as described later.

It can be expected for future hybrid systems that code and data are no longer stored in NOR-flash memory but at least parts of it end up in NAND-flash, which is available for the consumer electronics side anyway. If this is the case, secondary storage management by an MMU becomes essential for the automotive part also.

If MMUs are to be used by the automotive control part of hybrid systems three new requirements have to be considered:

Small protection blocks In a safety relevant system memory protection has to offer more than just the isolation between address spaces of different processes. Consider an example where an application shall be allowed to access the control registers of peripheral unit A in read-only mode and those of peripheral unit B not at all. In case of memory-mapped IO a memory protection unit could be used to achieve this, even if the control register blocks have adjacent addresses. An alternative solution is an underlying software layer (e. g. the operating system) that enforces protection with an arbitrarily small block size. However, this induces overhead and requires a larger trusted software base in case of a hybrid system.

High memory efficiency The market for classical automotive electronic control units is very cost sensitive. Even cost differences in the order of Cents per ECU can make a difference in such a mass volume market. As long as this is the case, the usage of memory has to be highly efficient in order to prevent higher hardware costs.

Real-time behavior Classical automotive software has to fulfill hard real-time constraints very often, e. g. to guarantee stability of control loop algorithms, or because of physical conditions, or safety requirements. New technologies often come with two problems regarding real-time requirements. First they often introduce timing deviations, i. e. depending on the execution history the same code might be executed faster or slower, even if the same data values are given. Consider for instance the case of an instruction cache. Second it might be very hard or even impossible to predict in a sufficiently precise way the worst-case execution time (WCET) for a given program, due to the unpredictable nature of a concrete mechanism causing timing deviations. When introducing a new technology such as MMUs the goal should be to add no further sources of timing deviations, if this is not completely possible the design of the technology should at least allow for a precise prediction of WCETs.

3 Prevailing MMU solutions

Standard IT systems such as personal computers or servers nowadays use page based MMU approaches. The logical address space is organized in pages of equal size, e. g. 4 kByte, or 1 MByte. Each process sees a logical address space of the size of the addressable area of the processor. The number of usable address bits according to the processor architecture, e. g. 32, or 64, determines the size of the address space, which is usually quite different to the actually available memory.

Whenever a process accesses memory it uses a logical address that is translated into a physical memory address before the actual memory cell can be accessed. The operating system manages the assignment of logical to physical addresses by keeping one *page table* per process that contains for each memory page the base address of the assigned physical *page frame*.

The size of the page table is determined by the size of the address space and the page size. In the DEC PDP-11, a system with 16 address bits and pages of 8 kByte was used, the page table had 8 entries which were kept in special registers [AS98]. In modern computers page tables are far too large to be kept in registers. The ARM Cortex-A8 has 32 address bits and a variable page size (4 KB, 16 KB, 64 KB, 1 MB, and 16 MB) [JLH12]. With pages of 4 KB the number of pages to capture in the page table is more than a million. A page table of this size cannot be kept in registers. The Intel Core i7 uses 48 logical address bits (36 for physical memory) and supports 4 KB pages [JLH12]. With each page table entry requiring one 64 bit word, this yields more than 200 TB just to store the page table. Such a page table cannot be stored in main memory but has to be paged itself. Concepts like multi-level paging and inverted page tables are used to address the page table size problem.

When the page table is too big to be held in registers, the problem occurs that each memory access would require another memory access just to read the corresponding page table entry from main memory. This would reduce the memory performance in an unacceptable way. Therefore, such systems typically use a hardware cache called *Translation Lookaside Buffer (TLB)* to store recently used page entries. Only if the TLB does not hold the corresponding page entry, the entry is read from the page table in main memory. This case is referred to as *soft miss*.

In personal computers and servers processes typically use more addresses than are actually available in main memory and the number of pages present in main memory is usually significantly smaller than the number of pages used during the lifetime of a process. Currently unused pages are stored in secondary memory. Whenever a process accesses a page not present in physical main memory the MMU issues an interrupt to the operating system. The OS loads the missing page from the secondary storage, e. g. a hard disk. This situation is referred to as *hard miss*. Because the space in main memory for pages is limited, another page has usually to be written back to secondary memory whenever the OS loads a new page from secondary memory into the corresponding page frame. This process is called *swapping*.

Because the complete address space is divided into pages of equal size, the swapping algorithm can select a page to be replaced in an arbitrary fashion, e. g. following a least

recently used, or round-robin policy, without fragmenting the main memory. However, because of the fixed size, memory unused within pages (since the code size of a program is no multiple of the page size) is lost. This is called *internal fragmentation*.

4 Design Parameter Page Size

This Section investigates how the page size should be chosen with respect to the three major requirements: protection granularity, memory efficiency, or real-time behavior.

4.1 Protection Granularity

Since access rights are always the same for all addresses within a page, the most flexible approach to protection would be pages containing only one memory word. Consider for instance the former example of memory mapped IO and adjacent control registers of different devices, e. g. the CAN controller and a Digital-to-Analog Converter (DAC). Assume further that a piece of code needs write access to the CAN controller but shall not be allowed to access the DAC. The address areas of the CAN controller and the DAC cannot be expected to be aligned with page frames that are usually larger than the combined address areas of both devices. Therefore, the page size would have to be very small to allow for small enough protection blocks.[2] A similar case would be common data areas for communication used by different processes to transfer small amounts of data, e. g. via the RTE (AUTOSAR Run Time Environment).

4.2 Memory Efficiency

4.2.1 Internal Fragmentation

A central issue regarding memory efficiency is the effect of internal fragmentation. The amount of memory wasted by internal fragmentation is proportional to the number of partly used pages and their size. As with protection granularity this observation calls for small pages. The following quantitative experiments demonstrate the impact of the design parameter page size on memory efficiency.

For the experiments an AUTOSAR compliant system is considered. The only addition is the assumption that memory management is supported and that each AUTOSAR Runnable has its own logical address space. The current AUTOSAR standard only supports memory protection and no management of virtual addresses. Moreover, tasks are the smallest runtime unit for which memory access rights can be defined currently. However, because

[2]Note that, even if the alignment between the physical addresses of the control register blocks fits well with different page frames (in general that would be per chance only), the frames have to be small enough to prevent applying wrong access rights to other neighboring address areas.

AUTOSAR software components are based on runnables rather than tasks, it is natural to assume a protection scheme on this level of granularity. Anyway, a system supporting memory protection on task level only, can always be used for protection on runnable level by assigning each runnable to its own task.

For the processor a 32-bit architecture is assumed and a MMU supporting page sizes of 256 Byte, 512 Byte, 1 KB, 2 KB, or 4 KB. Four different exemplary task sets (sets of runnables) are considered. Each task set consists of a number of tasks that use runnables with different characteristics (i. e. code size and data size). For convenience and to limit the space used by tables in the paper, each task uses runnables of the same type, i. e. in a task set with n runnables in m tasks, each task has $\frac{n}{m}$ runnables and there are $\frac{n}{m}$ different runnable types. Any runnable uses the minimal number of code and data pages depending on the page size. Note that, the examples do not consider the need of separate pages for stack areas, which every task usually requires, or for pages needed to protect IO or to share data areas. The actual loss of usable memory is therefore worse than the given numbers.

Task Set Tiny This task set has 8 tasks, each of which uses 6 runnables of the type shown in Table 1 and uses a total memory for code and data of 186560 Byte.

Table 1: Task Set Tiny

Runnable Type	Code Size	Data Size	No. of Pages per Page Size				
			256	512	1024	2048	4096
T1	400	480	4	2	2	2	2
T2	720	920	7	4	2	2	2
T3	1280	1680	12	7	4	2	2
T4	1800	2040	16	8	4	2	2
T5	4000	10000	56	28	14	7	4
T6	4400	10800	61	31	16	9	5

Task Set Small This task set has 16 tasks, each of which uses 6 runnables of the type shown in Table 2 and uses a total memory for code and data of 932992 Byte.

Table 2: Task Set Small

Runnable Type	Code Size	Data Size	No. of Pages per Page Size				
			256	512	1024	2048	4096
S1	1040	1920	13	7	4	2	2
S2	1600	2680	18	10	5	3	2
S3	2760	3920	27	14	7	4	2
S4	3240	7216	42	22	12	6	3
S5	4000	10312	57	29	15	8	4
S6	5800	13824	77	39	20	10	6

Task Set Medium This task set has 64 tasks, each of which uses 3 runnables of the type shown in Table 3 and uses a total memory for code and data of 974592 Byte.

Task Set Large This task set has 100 tasks, each of which uses 8 runnables of the type shown in Table 4 and uses a total memory for code and data of 3582000 Byte.

Figure 1 shows the percentage of memory loss due to internal fragmentation of the given task sets for different page sizes. As expected, small pages lead to less fragmentation loss.

Table 3: Task Set Medium

Runnable Type	Code Size	Data Size	No. of Pages per Page Size				
			256	512	1024	2048	4096
M1	980	1536	10	5	3	2	2
M2	1904	3416	22	11	6	3	2
M3	3192	4200	30	16	9	5	3

Table 4: Task Set Large

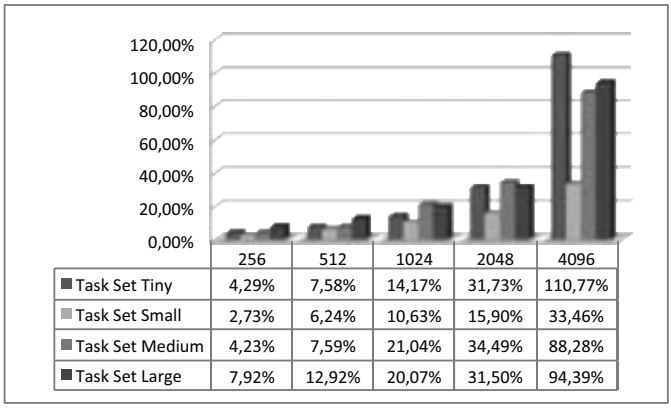
Runnable Type	Code Size	Data Size	No. of Pages per Page Size				
			256	512	1024	2048	4096
L1	1048	1904	13	7	4	2	2
L2	1556	1284	13	7	4	2	2
L3	2040	1360	14	7	4	2	2
L4	1612	1800	15	8	4	2	2
L5	1868	2572	19	10	5	3	2
L6	1560	1380	13	7	4	2	2
L7	2344	2616	21	11	6	4	2
L8	2756	8120	43	22	11	6	3

Task Set Small suffers from the least amount of fragmentation, but even there, more than 10 percent loss occurs as soon as the page size is 1 KB or above. A noteworthy observation is that the increase of the fragmentation rate differs substantially between task sets.

Under the assumption that these results are not significantly worse than in future automotive systems with memory management, a page size of 4 KB or above would certainly not be acceptable in practice. Given these numbers even 2 KB would be too much. However, it might still be the case that these numbers are misleading. One issue could be that a large amount of random sets would be quite different. To eliminate this uncertainty, Figures 2, 3, 4, and 5 compare the experimental results to the case that each last page is used by half only. The latter is to be expected for a large random experiment. As these figures show, the difference gives no reason to assume that the experiments leads to unjustified conclusions.

4.2.2 Page Table Size

Besides internal fragmentation the size of the page table itself is another important factor regarding memory efficiency. A small page size leads to many page table entries that need to be held in memory. Naturally, the page table is bigger for a larger address space. With 32 address bits 4 GB of memory can be addressed and the page table then has a size of 16 MB for a page size of 2 KB (assuming 8 bytes per page table entry). Of course the number of address bits and thereby the usable address space can be reduced. However, as Table 5 shows, especially for small page sizes this is still a significant factor, because every runnable with an own address space also needs its own page table and the OS will usually keep several page tables in main memory simultaneously. Consider that the amount of RAM in current ECUs is usually in the range of below 100 KB to at most 2.7 MB for the newly announced Infineon AURIX platform. Assume for instance for a future system a



	256	512	1024	2048	4096
Task Set Tiny	4,29%	7,58%	14,17%	31,73%	110,77%
Task Set Small	2,73%	6,24%	10,63%	15,90%	33,46%
Task Set Medium	4,23%	7,59%	21,04%	34,49%	88,28%
Task Set Large	7,92%	12,92%	20,07%	31,50%	94,39%

Figure 1: Unusable memory due to internal fragmentation

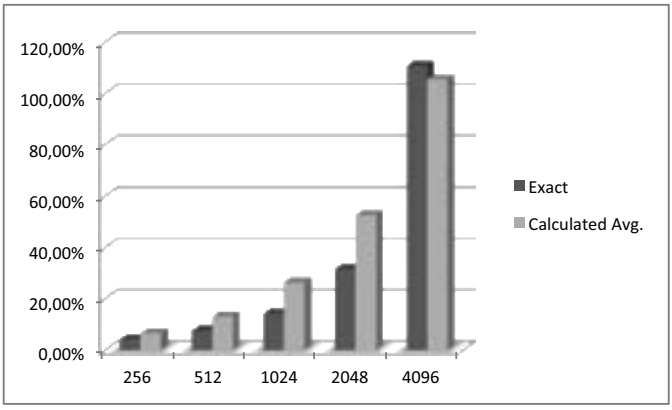


Figure 2: Task Set Tiny: comparison of experimental results and calculated average.

page size of 1 KB an address space of 16 MB and a total amount of 8 MB RAM and 10 tasks for which the page table has to be in memory simultaneously. In this scenario, the page tables would consume more than 15% of RAM in addition to the loss due to internal fragmentation.

4.3 Performance and Real-Time Behavior

The design of an MMU can influence performance and real-time behavior of future automotive systems in more than one way. As discussed above, the size of the pages determines the number of page table entries. With too many entries, a page table cannot be held in an array of registers but has to be placed in main memory. For each memory access this would require another memory access to look up the proper page table entry.

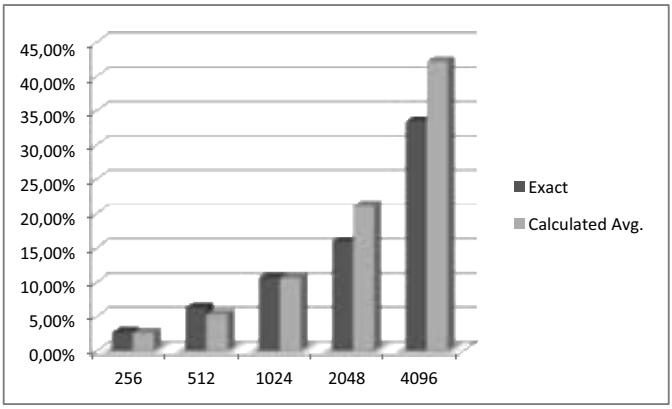


Figure 3: Task Set Small: comparison of experimental results and calculated average.

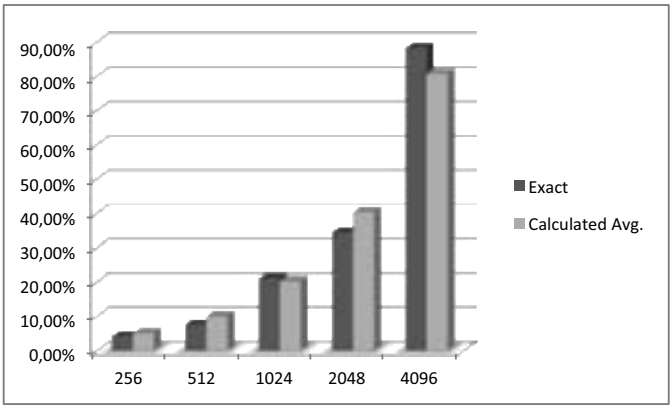


Figure 4: Task Set Medium: comparison of experimental results and calculated average.

This dramatic performance penalty is usually reduced by caching the page table in a fully associative register set called Translation Look-aside Buffer (TLB). The effect of such a solution and further issues regarding real-time behavior are presented below.

4.3.1 Immediate Influence on Program Level

This Section considers the immediate effects on program execution time and how to predict these. Indirect effects, e. g. late effects (i. e. effects that occur at later program points), are discussed later.

Address Translation Introducing MMUs requires to translate logical addresses into physical addresses for each memory access. As explained above, adding another memory access to read from a page table in memory is, although it would prevent timing deviations, not acceptable because of its serious performance impact.

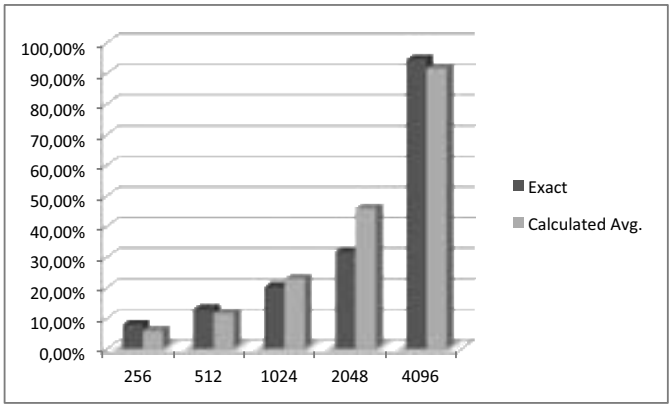


Figure 5: Task Set Large: comparison of experimental results and calculated average.

Table 5: Memory demand of page tables

Page Size	**Page Table Size per Address Space Size [KB]**					
	4 GB	**1 GB**	**256 MB**	**64 MB**	**16 MB**	**4 MB**
256	131072	32768	8192	2048	512	128
512	65536	16384	4096	1024	256	64
1024	32768	8192	2048	512	128	32
2048	16384	4096	1024	256	64	16
4096	8192	2048	512	128	32	8

Another approach to prevent timing deviations is to keep the translation table in processor registers. This is only viable, if the number of table entries is reasonably small. From a real-time perspective this is clearly the best approach as it is perfectly predictable and introduces no timing deviations.

The usual approach in standard IT systems is to use TLBs. If it can be guaranteed that a translation entry is always present in a TLB when needed, such a system can be designed to have no (additional) timing deviations. To achieve this, TLB entries could be loaded by the RTOS (Real-Time Operating System) into TLBs before a task needs it.

If the situation of TLB misses (also called soft misses) cannot be avoided completely, it might still be possible to eliminate it for certain parts of the system. This could for example be achieved by partial locking of TLBs. This feature is already available in some hardware architectures.

When a TLB miss occurs in a system with hardware managed TLBs the execution time of the concerned instruction is immediately increased by the memory access to the page table. It is important to consider this when analyzing the worst case execution time (WCET) [WEE+08] of a runnable or task. Depending on the hardware architecture this delay can be (partially) compensated by a pipelined execution. This effect can be considered when using an integrated cache and pipeline analysis as described in [Sch00]. As long as the hardware exhibits no timing anomalies [LS99] it is sufficient, but pessimistic,

to consider this immediate delay as TLB miss penalty outside of the WCET analysis.

The accuracy of a TLB miss prediction will depend on the policy to select TLB entries for replacement in a similar way as for cache misses. The best results can be expected for *LRU* (least recently used) algorithms, because of the similarity to caches [RGBW07]. For performance reasons a hardware implemented replacement policy is usually preferable. However, if a certain hardware architecture offers no LRU policy, a software based mechanism might be better regarding temporal predictability.

In case of a software managed TLB an interrupt is issued to the CPU and the RTOS has to handle the updating of the TLB. As opposed to hardware managed TLBs the current instruction is interrupted and a context switch into the RTOS is performed. After the RTOS has updated the concerned TLB it returns to the interrupted instruction and the TLB lookup is repeated. This is directly comparable to asynchronous interrupts by the RTOS for other reasons and suitable approaches to consider such cases can be found in [Sch02, Sch00, Sch03].

Protection When a memory access violates the protection settings monitored by the MMU, an interrupt is issued to the CPU and the OS has to handle this exception. Regarding immediate influence on real-time behavior this is not different to the above mentioned case of interrupts due to TLB misses in systems with software managed TLBs. Since a protection violation is no expected system condition it would be necessary to find an upper bound on the frequency of such errors to predict whether a system satisfies its real-time requirements. However, this problem occurs alike with systems using memory protection units (MPUs).

Managing the Translation Table Setting up the translation table for each runtime unit (e. g. an AUTOSAR runnable) takes additional time that has to be considered when reasoning about real-time guarantees. Depending on the way this is handled in a particular system, this can range from one time initialization at startup to recreating the table after each preemption. Considering the resulting impact on the real-time behavior requires no fundamentally new methods but special support by tools is needed.

4.3.2 Indirect Influence on Program Level

In this Section late effects (a late effect is a situation where an initial cause leads to an increase in execution time at a distant place, i. e. at a later point in time) and other indirect influence on program level is considered.

Besides the replacement of entries in the TLB itself further cases have to be considered. When a TLB miss occurs at least the corresponding translation table entry has to be read from memory. In systems with hardware managed TLBs the MMU itself performs this read operation. Unless the hardware designers undertake special measures to prevent this, the memory access involves updating the cache state and therefore potentially influences the later execution of programs. In the case of software managed TLBs additional memory accesses to fetch the associated RTOS function and involved data are necessary and change the cache state. The potential influence within the same or other programs has to be considered. Suitable approaches that could serve as solution pattern can be found

in [Sch00, Sch03]. Note that in multicore systems the state of shared caches can also be affected.

Another late effect can occur in systems with a write back cache policy. If cache content is displaced that needs to be written back to main memory, it could happen that the matching translation table entry is no longer in the TLB. Thus the write back could cause later TLB misses and these TLB misses could cause cache misses. Considering these cases would introduce significant loss of precision for analysis tools. In general data caches in real-time systems should not be configured to use a write back policy.

In systems with branch prediction unexpected delays can occur if a branch is mispredicted and the CPU fetches the wrong target [RWT+06]. Since this can only happen if the wrong target is in the cache, the situation is counter intuitive (a cache hit leads to longer execution times than a cache miss). Such counter intuitive behavior is named a timing anomaly. In similar manner a TLB hit (in conjunction with a cache hit) could lead to a higher execution time.

4.3.3 Immediate Influence on Scheduling Level

This Section discusses increased context switch costs and other immediate effects at scheduling level.

As mentioned before, TLB misses in case of software managed TLBs cause interruptions of the current code. The system then switches to RTOS context, executes the appropriate RTOS function, and switches back to user level. This sequence has to be considered when analyzing the real-time behavior of a system.

Furthermore, it has to be considered generally that a context switch usually involves changing the page table. If TLBs are used this might involve flushing TLBs. In hardware architectures that share TLBs between processes (runtime units) flushing of TLBs is not necessary. But subsequent TLB misses can still occur due to the context switch.

If TLB entries belonging to one process cannot be affected by other processes, the prediction of the temporal behavior of a process regarding TLB misses can be performed on the associated program alone. Otherwise, the scheduling of processes has to be considered, similar to caches the program level and scheduling level analysis have to be integrated more closely in such a case, cf. [Sch00].

Protection When a memory access violates the protection settings monitored by the MMU, an interrupt is issued to the CPU and the OS has to handle this exception. Regarding real-time behavior this is not different to the case of using a memory protection unit. Therefore, this issue is not further elaborated here, even though it is not trivial from a systems perspective.

4.3.4 Indirect Influence on Scheduling Level

Using the MMU to transfer code or data into RAM Although this feature is out of the scope of this paper, it should be briefly mentioned that using MMUs to transfer code or

data into RAM induces serious impact on real-time behavior in several ways.

Consider a system where code has to be loaded into RAM before being executed.[3] Whenever a requested page is not present in RAM it has to be loaded from secondary memory. Such a situation is called a *hard miss*.

Either the RTOS immediately loads the missing page from the secondary storage or the current task is blocked to favor another task that can run immediately. In the first case the WCET of the concerned task is artificially increased thus reducing the possible processor load. In the second case the scheduling method has to support blocking of tasks in the middle of their execution. The AUTOSAR standard does not support this and serious changes on the OS level would be necessary to allow for this. For instance, while Tasks in AUTOSAR can wait for events, they have to explicitly call the special system call *WaitEvent()*. In addition tasks may never hold a resource whenever they call *WaitEvent()*. The reason is that the OSEK/AUTOSAR mechanism (Immediate Ceiling Priority Protocol) for mutual exclusion uses priorities to ensure its properties such as absence of deadlocks and predictability of worst case behavior. If a task could be blocked in middle of execution the mechanism could no longer be used.

5 Conclusion

The issues discussed in this paper demonstrate that the prevailing page based MMU approaches have serious drawbacks from the perspective of automotive software engineering. The achievable protection granularity is inferior to memory protection units. In terms of memory efficiency neither large nor small pages lead to acceptable results, as large pages cause a lot of internal fragmentation and small pages lead to serious memory demand by page tables (cf. Table 5). The performance suffers from the need for page table management and the real-time behavior is hard to predict in presence of TLBs and due to other issues.

Still the wish for a hardware supported memory management in automotive systems will probably grow into a nonrejectable requirement over time. Fortunately, there is a long known approach that offers almost anything the automotive industry will need here: a small and variable protection granularity, freedom from internal fragmentation, translation tables that fit into register arrays, and as a consequence no problems to predict the real-time behavior. All this could be provided with a *segmentation based MMU*.[4] In standard IT systems the segmentation based MMU approach is at the border to extinction. Probably this is the case because many people see swapping as the primary purpose of MMUs and the fact that segments of varying size require sophisticated algorithms in this regard other than pages of uniform size that can be replaced in an arbitrary fashion.

May be we will see a renaissance of segmentation based MMUs due to their advantages in the automotive domain and fields with similar requirements such as avionics. Whether this will be the case or not remains to be seen.

[3] This is for instance the case when a system uses NAND Flash as secondary memory.

[4] E. g. 32 size adjustable segments per runnable would require 64 registers plus some logics.

References

[ari97] Avionics Application Software Standard Interface, 1 1997. ARINC Report 653.

[AS98] Peter Baer Galvin Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley, 1998.

[CRA+09] Joao Craveiro, Jose Rufino, Carlos Almeida, Rui Covelo, and Pedro Venda. Embedded Linux in a partitioned architecture for aerospace applications. *Computer Systems and Applications, ACS/IEEE International Conference on*, 0:132–138, 2009.

[GWLC12] Zonghua Gu, Zhu Wang, Shijian Li, and Haibin Cai. Design and Implementation of an Automotive Telematics Gateway Based on Virtualization. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*, pages 53 –58, april 2012.

[HH08] André Hergenhan and Gernot Heiser. Operating Systems Technology for Converged ECUs. In *6th Embedded Security in Cars Conference (escar)*, Hamburg, Germany, Nov 2008. ISITS.

[JLH12] David A. Patterson John L. Hennessy. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann, 2012.

[LS99] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *WCET*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Sch00] Jörn Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *21st IEEE Real-Time Systems Symposium*, pages 195–204, November 2000.

[Sch02] Jörn Schneider. Why You Can't Analyze RTOSs without Considering Applications and Vice Versa. In *2nd International Workshop on Worst-Case Execution Time Analysis*, Vienna, Austria, June 2002.

[Sch03] Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universität des Saarlandes, 2003.

[Sch12] Jörn Schneider. Overcoming the Interoperability Barrier in Mixed-Criticality Systems. In *19th ISPE International Conference on Concurrent Engineering–CE2012*. International Society for Productivity Enhancement, September 2012. To appear.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

SW-basierte Integration von neuen Fahrzeugfunktionen in zentralisierten Controllern

Klaus Becker[1], Christian Buckl[1], Alexander Camek[1], Rainer Falk[2], Ludger Fiege[2], Jürgen Geßner[2], Stephan Sommer[1]
1) fortiss gemeinnützige GmbH, Guerickestraße 25, 80805 München
2) Siemens AG, Corporate Research and Technologies,
Otto-Hahn-Ring 6, 81730 München

Abstract: In aktuellen Fahrzeugen wird ein wesentlicher Teil der Fahrzeugfunktionen durch Software realisiert. Die Integration aktiv eingreifender Assistenzsysteme wird diesen Trend noch verstärken und die Komplexität des Bordnetzes wird weiter zunehmen. In diesem Artikel stellen wir einen Ansatz vor, Bordnetz und Software im Fahrzeug über eine datenzentrische Middleware zu entkoppeln. Sie koordiniert die Kommunikation zwischen Funktionen, Sensorik und Aktorik zur Laufzeit. Basierende auf einer Zentralrechnerarchitektur wird eine redundante Datenbasis zur Verfügung gestellt, die eine fail-operational Ausführung auch von sicherheitskritischen Funktionen erlaubt. Wir stellen am Beispiel Plug-and-Play (PnP) eine neue Sekundärfunktion vor, die durch diesen Ansatz ermöglicht wird. Safety- und Security-Aspekte der vorgestellten Architektur werden ebenfalls betrachtet.

1. Einführung und Motivation

Heutige Automobile sind ein komplexes Zusammenspiel von verteilen, gekoppelten Einzelsystem. Derzeitige Premiumfahrzeuge enthalten eine hohe Anzahl von bis zu ca. 100 Steuergeräten (Electronic Control Units, ECUs), die mit verschiedenen Bussystemen miteinander verbunden sind. Selbst wenn Funktionen in *Domänenkontrollern* konsolidiert werden, so ist der entstehende Verkabelungsaufwand jedoch enorm. Assistenz- und Komfortsysteme, die in den Antriebsstrang eingreifen, machen die Integration im Hinblick auf die funktionale Sicherheit sehr aufwändig.

Ein weiterer Treiber der Domänen übergreifenden Integration sind hochintegrierte Radnaben- oder radnahe Elektromotoren. Die Integration von Lenkung, Vortrieb, Verzögerung wie auch von ABS und ASR in ein einziges, autark arbeitendes Modul, lässt Differenzial, Torque-Vectoring sowie die Beschleunigung selber zu einer verteilten Steuerung werden, in der die Funktionen der Module zentral koordiniert werden müssen. Als Antwort sind in Avionik, Industrieautomatisierung und Robotik zentralisierte Architekturen entworfen worden, die auch bereits in automobilen Forschungsprojekten angewendet wurden (vgl. [SP02] und [HA11]). Dies wäre eine technische Konsolidierung wie sie zum Beispiel mit dem Wechsel von diskreten Datenleitungen hin zu Bussystemen bereits erfolgte (vgl. Abbildung 1).

In diesem Artikel stellen wir eine datenzentrische Middleware vor, die die Entwicklung neuer Funktionen und die Ankopplung der Sensorik/Aktorik über ein Backbone-

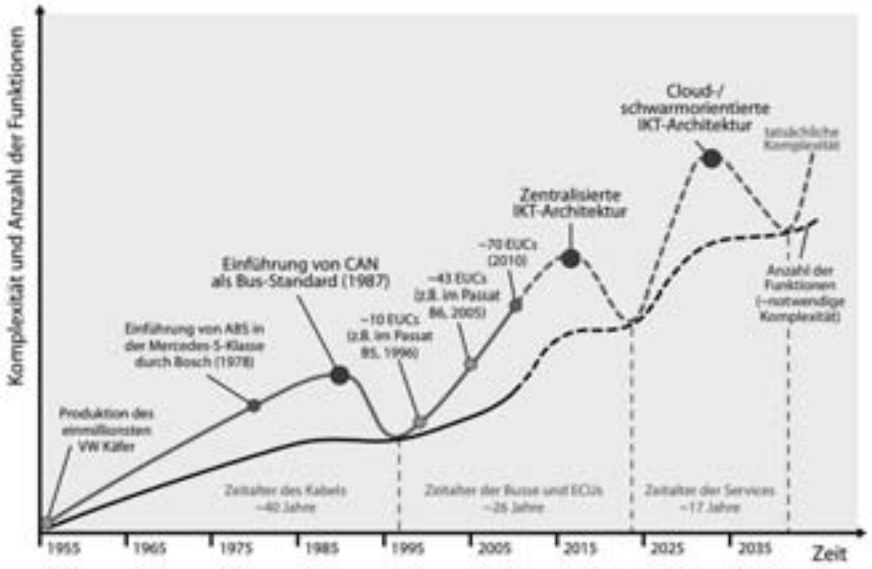


Abbildung 1: Wachsende Komplexität durch inkrementelle Weiterentwicklung [Fo12]

Netzwerk entkoppelt (vgl. [Bu12], [Br11]). Basierend auf vorverarbeitender, intelligenter Peripherie, werden Daten in aufbereiteter Form in einer Datenbank abgelegt. Die Daten werden allen Steuergeräte/SW-Funktionen zur Verfügung gestellt, die dafür konfiguriert wurden. Als wesentlicher Treiber wird die Integration neuer Funktionen auch nach Auslieferung des Fahrzeugs betrachtet, um damit Plug-and-Play Szenarien zu unterstützen. Die Middleware muss dabei die Ressourcen-Zuweisung und Safety-bzw. Security-Anforderungen umsetzen. Die Validierung der Konzepte wird im Rahmen des BMWi Förderprojektes „Robust and Reliable Automotive Computing Environment for Future eCars“ (RACE) erfolgen [RA12]. Anhand von zwei Versuchsfahrzeugen werden die technischen Grundlagen umgesetzt und evaluiert werden.

Dieser Artikel ist wie folgt strukturiert: Kapitel 2 stellt zunächst die zugrunde liegende zentralisierte Systemarchitektur vor und vergleicht diese mit existierenden Ansätzen. In Kapitel 3 wird der datenzentrische Ansatz vorgestellt. Wichtige Systemaspekte, wie Plug-and-Play, Safety und Security, werden in Kapitel 4 beschrieben. Kapitel 5 schließt mit einer Zusammenfassung.

2. Zentralisierte Systemarchitektur

Ein Kern der Systemarchitektur, die sich hier an Avionik und Industrieautomatisierung orientiert, ist eine deutlich horizontale Schichtung der Architektur. Die heute vorherrschende Bündelung von Funktion, ECU, Sensorik in Teilsystemen wird ersetzt von einer Aufteilung in Feldebene zur Anbindung von Sensorik/Aktorik, lokalen Regelung von

Stellgrößen (Motoren) und koordinierenden bzw. planenden Aufgaben in der Steuerungs- bzw. Leitebene.

Die betrachtete Systemarchitektur umfasst zentrale Rechner für koordinierende Aufgaben, ein Ethernet-basiertes Backbone und intelligente Sensoren/Aktoren, die direkt oder via Sektor-Gateways (z.B. für den Frontbereich) angeschlossen sind. Grundsätzliches Merkmal der zentralen Rechner ist in allen Fällen, dass sie keine funktionsspezifischen Ein-/Ausgaben (ADC, spezielle Busse, etc.) besitzen, sondern lediglich an das Backbone-Netzwerk angeschlossen sind.

Die Zentralrechner übernehmen das vorausschauende Planen und setzen zusätzlich modulübergreifende, überlagerte Regelungen und koordinierende Funktionen um [Sp02]. Sie bilden beispielsweise den Fahrvektor aus den Vorgaben des Fahrers und der Assistenzfunktionen und setzen diese in Leistungsanforderungen (Zielwerte) an die Motoren (Aktoren) um. Die Radmodule bestehen aus E-Motoren, Inverter und lokaler Regelung, die schnelle, geschlossene Regelungsaufgaben übernimmt und die Zielwerte von den Zentralrechnern erhält. In Zukunft können die Radmodule auch Grundfunktionen der Blockier- und Schlupfregelung übernehmen, jedoch enthalten diese Module prinzipiell keine koordinierenden Aufgaben zwischen anderen Aktoren bzw. Aufgaben, die nichts mit ihrer lokalen Kontrolle physikalischer Größen zu tun haben.

Die hohe Integration eines Elektroradmoduls lässt sich hervorragend mit einer X-by-Wire Architektur kombinieren. Eine beispielhafte Darstellung für ein X-by-Wire-Fahrzeug ist in Abbildung 2 dargestellt.

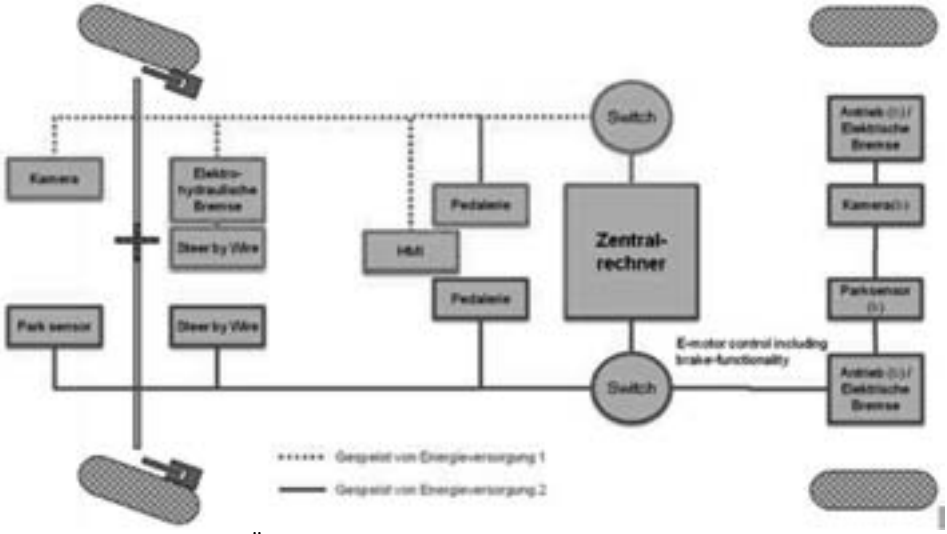


Abbildung 2: Übersicht Systemarchitektur mit logisch zentralem Rechner

Da es für ein fahrendes Automobil keinen sicheren Zustand gibt, müssen sicherheitskritische Funktionen, wie z.B. Lenkung und Bremse, ASIL-D Anforderungen erfüllen. Dies betrifft sowohl die Kommunikation als auch die Zentralrechner und die Energieversorgung. Gemäß der ISO 26262 Anforderungen muss die Wahrscheinlichkeit eines fehlerfreien Betriebs bei Einzelfehlern bei >99% bzw. für Mehrfachfehler bei > 90% liegen und das bei einer Fehlerrate von weniger als 10^{-8} /h.

Es gibt mehrere Architekturkonzepte, die ASIL-D Anforderungen mit Fail-Operational-Verhalten erreichen: Wir wählen einen Duo-Duplex-Aufbau. Jeweils ein Paar zentraler Rechner ist in einem Duplex Control Computer (DCC) verbunden, um die Ein/Ausgangsdaten gegenseitig zu überwachen und Inkonsistenzen festzustellen. Ein zweites Paar (zweiter DCC) läuft als hot-standby, um im Fehlerfall die Funktionalität aufrecht zu erhalten. Der Duo-Duplex-Ansatz lässt sich im Vergleich zu Triplex-Systemen leichter skalieren und die Aktoren benötigen kein systemisches Wissen und keine an den Grad der Redundanz gebundenen „Voter“ zur Fehlererkennung.

Das Netzwerk muss ebenfalls redundant aufgebaut werden. Hier kommt eine Ringtopologie basierend auf Ethernet zum Einsatz; für Details sei auf [Ar12] verwiesen.

Aus den Verantwortlichkeiten der Sensoren/Aktoren und der Zentralrechner (DCC) ergibt sich, dass Zustands- und Sensordaten über das Netzwerk den Applikationen in den DCCs zur Verfügung gestellt werden müssen. Umgekehrt müssen die errechneten Zielwerte den Aktoren übermittelt werden. Als Zykluszeit für den regelmäßigen Austausch dieser Daten ergibt sich ein Richtwert aus den Erkenntnissen der Projekte Sparc und HAVEit. Dort waren zentralisierte, koordinierende Funktionen mit einer Zykluszeit im Bereich von 10-20ms für betrachtete Fahrsituationen eines PKW ausreichend.

2.1. Relation zu AUTOSAR

Die hier vorgestellten Konzepte sind kein Ersatz für AUTOSAR, sondern konzentrieren sich auf die Kombination von Erweiterbarkeit (zur Laufzeit) und Safety Mechanismen, die die Entwicklung von Applikations- und Sicherungsfunktionalität trennen. Die Konzepte sind zunächst orthogonal zu AUTOSAR und könnten daher zukünftig als eine mögliche Erweiterung angesehen werden.

Die vorgestellte Architektur schreibt systemische Redundanzen vor, die so im AUTOSAR Standard nicht gefordert werden. Dies erlaubt zum einen Sicherungsmechanismen zu entwerfen, die HW-Fehler unabhängig von den Applikationen erkennen und behandeln. Dies beinhaltet auch den mixed-criticality Betrieb, in dem Anwendungen unterschiedlicher Safety-Levels auf einer DCC laufen und kommunizieren können. Des Weiteren wird die datenzentrische Middleware oberhalb der AUTOSAR RTE-Funktionalität Datenfusion und -bereitstellung anbieten, die nicht statisch konfiguriert sondern zur Laufzeit geändert werden kann (s. Abschnitt 4.1). Im Rahmen der laufenden Arbeiten vermeiden wir zunächst den Aufwand, AUTOSAR in einer mixed-criticality Umgebung einzusetzen bzw. mit einer dynamischen Middleware zu verbinden.

3. Datenzentrische Middleware

Die datenzentrische Middleware muss also die Elemente der Systemarchitektur (Sensoren, Aktoren, DCCs) verbinden und dabei die folgenden Ziele umsetzen: Echtzeitfähigkeit (Zykluszeit 10ms), Erweiterbarkeit, Mixed-criticality (Kombination QM bis ASIL-D, fail-operational), sowie Security. Im Gegensatz zu den aktuell verfolgten Ansätzen, bei denen von verschiedenen Steuergeräten Nachrichten verarbeitet werden, die auf dem Bus anliegen, wird bei dem hier vorgestellten Ansatz mit Daten gearbeitet, die ein Abbild des aktuellen Fahrzeugzustands wiederspiegeln.

3.1. Übersicht der datenzentrischen Architektur

Zur Entkoppelung der Anwendungen von der Infrastruktur (Netzwerk, ECU) werden in der datenzentrischen Architektur vorab verschiedene Datentypen (so genannte Topics) definiert. Beispiele für Topics sind die Fahrzeuggeschwindigkeit, Radgeschwindigkeit und Außentemperatur. Softwarekomponenten kommunizieren miteinander ausschließlich über an Topics gebundene Daten, indem sie diese an die Middleware weitergeben bzw. diese von der Middleware erhalten. Neben den Topics kann es notwendig sein, die Anforderungen an die Daten noch näher zu beschreiben. Dazu werden Meta-Daten verwendet. Diese beschreiben sowohl Anforderungen an die Daten als auch Garantien. Beispiele für Metadaten sind Ort der Erzeugung, Genauigkeit, Konfidenz und Sicherheitslevel.

Die Ausführung der Anwendungen erfolgt zyklenbasiert (Zykluslänge z.B. 10ms). Dabei werden die Daten zum Beginn des Zyklus von der Middleware in den Arbeitsbereich der in diesem Zyklus startenden SW-Komponenten kopiert und bei Beendigung der Funktion aus logischer Sicht (siehe Ausführungszeiten [He01]) wieder in die Middleware zurückkopiert. Ein Zugriff der Funktion auf weitere Daten während der Abarbeitungszeit ist nicht möglich. Neben volatilen Daten, also Daten, die in jedem Zyklus neu erzeugt werden, sollten auch persistente Daten unterstützt werden. Persistente Daten behalten dabei solange ihren Wert, bis ein neuer Wert vorliegt.

Um die Verschaltung der Software-Komponenten zu ermitteln geben diese an, welche Daten sie benötigen und welche sie bereitstellen. Eigenschaften wie Latenz und Jitter werden dabei durch Quality of Service (QoS) Parameter beschrieben und fließen in die Berechnung mit ein. Auf Basis dieser Informationen wird schließlich die Verschaltung der Software-Komponenten berechnet. Als Basis der Umsetzung in RACE wird die Middleware Chromosome [CH12] verwendet und im Zuge des Projektes erweitert.

3.2. Chromosome (XME)

Die Chromosome-Middleware (siehe Abbildung 3) besteht aus mehreren, modular gestalteten Komponenten. Eine Auswahl dieser Komponenten wird in diesem Abschnitt betrachtet.

Die Hardware-Basis stellen die in Kapitel 2 vorgestellten, gedoppelten Rechner eines DCCs dar. Zur Unterstützung von Anwendungen verschiedener Kritikalitäten wird in RACE ein Betriebssystem mit Virtualisierungsunterstützung verwendet, die Portierbarkeit auf neue Plattformen wird durch einen Hardware Abstraction Layer (HAL) erleichtert. Dieser kapselt den Zugriff auf die Hardware bzw. das Betriebssystem und setzt die Arbitrierung beim gleichzeitigen Zugriff auf Ressourcen um.

Die eigentlichen Middleware-Funktionen werden dann im oberen Teil (in der Abbildung hellblau gekennzeichnet) umgesetzt. Die Kommunikation ist im Interface-Manager gekapselt, der, je nach Anforderung der Anwendung, mit Plug-Ins für verschiedene Kommunikationsstandards versehen werden kann. In Standardanwendungen sind dies häufig TCP und UDP, in RACE wird Ethernet (Layer 2) für die sicherheitskritische Kommunikation verwendet. Empfangene Daten werden in der Middleware (Datenbank, siehe nächster Abschnitt) zwischengespeichert, vom I/O Management validiert und anschließend über den Broker an die einzelnen, lokalen Empfänger verteilt. Dies erlaubt eine vollständige Entkoppelung von Sender (Daten-Produzent) und Empfänger (Daten-Konsument). Die Verschaltung der Komponenten wird dabei vom Directory berechnet und als Konfiguration an die Broker verteilt. Die jeweilige Ausführung der Komponenten (sowohl der Middleware als auch der Anwendung) wird vom Execution Manager angestoßen und überwacht.

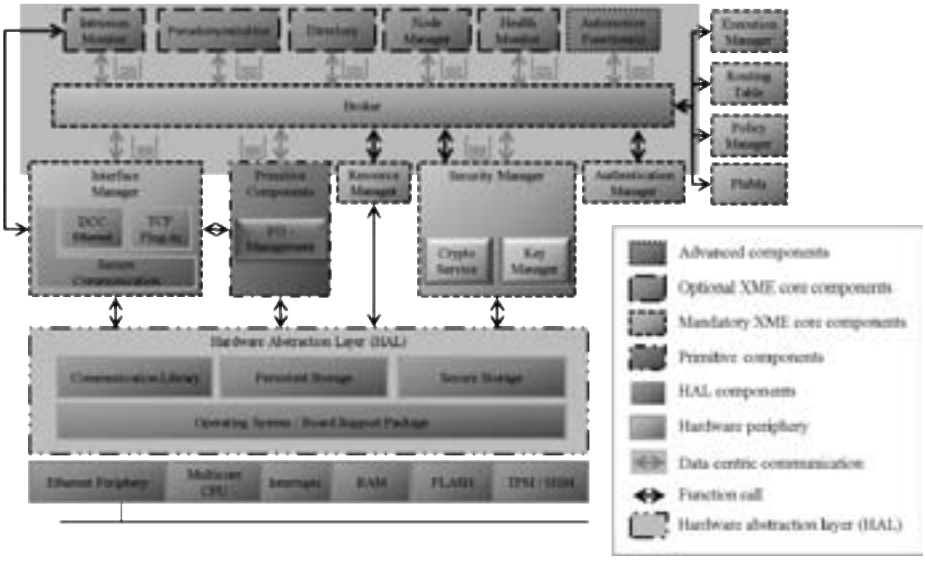


Abbildung 3: Chromosome Stack-Übersicht

Um die Echtzeitfähigkeit des Systems zu garantieren, wird die Ausführung der Funktionen offline geplant und online überwacht. Basierend auf Informationen unter anderem zur Worst-Case Execution Time (WCET) können dabei die Komponenten optimal eingeplant werden. Ebenfalls können dadurch Degradationsstrategien festgelegt und hinterlegt werden. Die offline berechnete Allokation der Ressourcen und die Ausführungspläne für die verschiedenen Betriebszustände (incl. Degradationsstufen) sind im Plattform-Management hinterlegt. Zur Laufzeit wird durch das Plattform-Management der aktuelle Ausführungsplan im Execution Manager hinterlegt. Sobald ein Fehler erkannt wird, kann durch das Plattform-Management eine geeignete Alternativkonfiguration im Execution Manager hinterlegt werden. Für nicht sicherheitskritische Anwendungen ist die Ressourcenplanung für neue, vorher nicht bekannte Anwendungen zur Laufzeit möglich. Die Separation von den sicherheits-kritischen Funktionen wird dabei durch Virtualisierung des zugrunde-liegenden Systems und der dadurch erreichten Separation sichergestellt. Dies ermöglicht eine wesentlich höhere Flexibilität und somit das nachträgliche einbringen neuer Funktionen in das System (z.B. Plug-and-Play).

3.3. Logisch zentrale Datenbasis

Um den Anforderungen sicherheitskritischer Anwendungen gerecht zu werden, müssen Daten vor der Verarbeitung in den einzelnen Anwendungen auf Plausibilität geprüft werden. Als zentraler Punkt in XME dient hierzu eine logische, zentrale Datenbank als Datenbasis. Alle von XME bzw. den Anwendungen auf einem Knoten (DCC) verarbeiteten Daten werden zunächst in der logischen Datenbank zwischengespeichert. Zu Beginn eines Zyklus werden die Daten der Anwendung zur Verfügung gestellt. Die Ergebnisse der Anwendung werden nach der Berechnung zum Ende eines Zyklus wieder in die Datenbank geschrieben.

Bei sicherheitskritischen Anwendungen können die Berechnungen einer Anwendung dupliziert auf mehreren Knoten (DCCs) im Netzwerk ausgeführt werden. Dabei werden sowohl die Eingabedaten als auch die Ergebnisse der Berechnungen basierend auf den Ergebnissen in der Datenbank durch das I/O Management verglichen und bewertet. Neben den Aufgaben im Regelbetrieb ist die Datenbank ebenfalls von zentraler Bedeutung während der Entwicklung und der Systemtests. Dabei werden über die Datenbank synthetische Daten eingespeist bzw. Ergebnisse ausgelesen. Dadurch können künstlich Fehler provoziert bzw. ins System eingespeist und die Reaktion des Systems darauf getestet und aufgezeichnet werden.

3.4. Kontrollfluss

Wie bereits in Abschnitt 3.1 eingeführt, erfolgt die Ausführung von Funktionen in der RACE-Architektur Zyklus-basiert. Je nach Berechnungsaufwand können Anwendungen mindestens einen oder auch mehrere Zyklen zur Berechnung benötigen. Dabei werden der Anwendung die Daten zu Beginn des ersten Zyklus' von der Datenbank zur Verfügung gestellt und am Ende der Berechnung wieder zurück an die Datenbank

übertragen. Nach einer optionalen Konsistenzprüfung stehen die Daten dann im nächsten Zyklus für weitere Komponenten zur Verfügung.

Eine beispielhafte Anwendung ist in Abbildung 4 dargestellt. Weder die Sensoren noch die Aktoren sind dediziert für die Funktion Notbremsassistent installiert, noch direkt mit einem dedizierten Steuergerät verbunden, das exakt diese Funktion abbildet. Alle beteiligten Softwarekomponenten werden auf einem Zentralrechner (DCC) ausgeführt.

Zu Beginn werden die Messwerte der Sensoren zuerst über eine Komponente zur Datenfusion vorverarbeitet und in Zusammenhang gesetzt. Die Ergebnisse, wie z. B. Objektlisten, werden dann an die Software-Komponente des Notbremsassistenten übertragen. Im Arbitrierer werden die Ergebnisse des Notbremsassistenten mit den Eingaben des Fahrers abgeglichen und die Entscheidung über die Fahrstrategie berechnet. Über die Softwarekomponente des Antriebsstrangs werden im Anschluss die einzelnen Stellgrößen für die Aktoren ermittelt und an diese übertragen. In dem gezeigten Anwendungsfall sind die Aktoren zugleich Sensoren. Sie kennen den eigenen Zustand und ermöglichen so der Softwarekomponente für den Antriebsstrang einen Soll / Ist Abgleich.

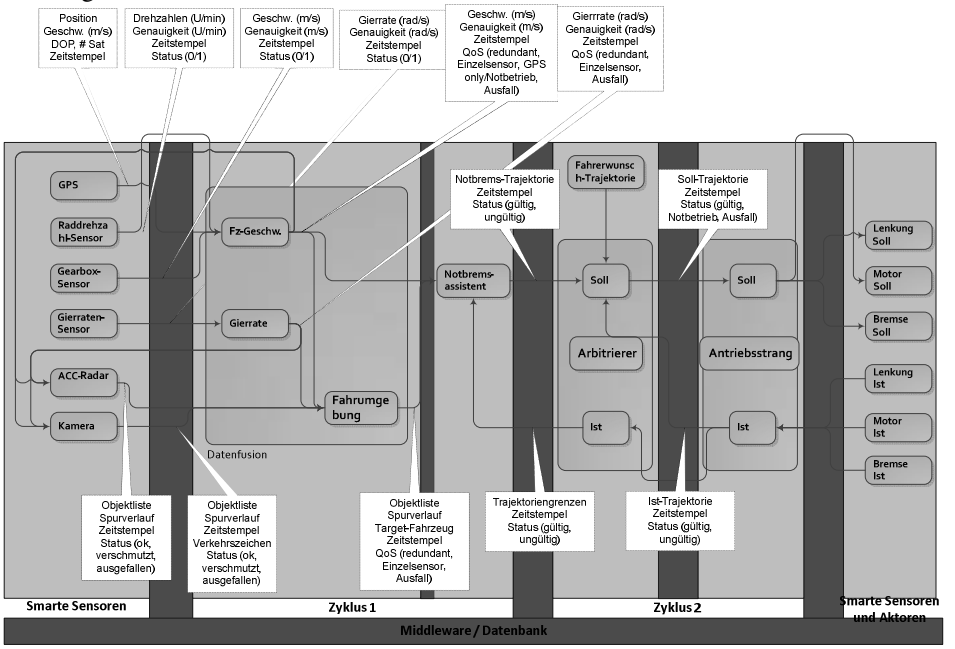


Abbildung 4: Funktionsblöcke der Anwendung Notbremsassistent

Durch die Entkoppelung der einzelnen Funktionen und die Verfügbarkeit der Informationen im System wird im Gegensatz zur Realisierung in einem dedizierten Steuergerät die Möglichkeit geschaffen, die Informationen für weitere Funktionen zu nutzen. Dies ermöglicht es den Herstellern z. B. bei bereits ausgelieferten Fahrzeugen Sicherheitsfunktionen nachzurüsten, da nur eine weitere Software-Komponente

installiert werden muss. Für nicht sicherheitskritische Funktionen ist sogar eine Installation von neuen Anwendungen durch den Benutzer während der Fahrt möglich.

4. Im Fokus stehende Systemaspekte

In diesem Kapitel beschreiben wir wichtigen Sekundärfunktionen, die mit Hilfe der vorgestellten Middleware umgesetzt werden.

4.1. Nachträgliches Hinzufügen neuer Funktionen (Plug-and-Play (PnP))

Das vorab beschriebene Zentralrechnerkonzept und die Datenzentrik sind eine wesentliche Grundlage, um ein nachträgliches (After-Sale) Hinzufügen von neuen Fahrzeugfunktionen und neuer Fahrzeughardware (z.B. Sensorik) per PnP zu ermöglichen. Bezüglich bereits existierender und neu hinzuzufügender sicherheitskritischer Anwendungen muss das PnP-Konzept besonderen Qualitätsmerkmalen genügen, um die Safety-Eigenschaften nicht negativ zu beeinflussen.

Warum PnP? Eine wesentliche Beschaffenheit von Heimcomputern ist die individuelle nachträgliche Erweiterbarkeit um neue Funktionalitäten in Form von neu installierten Programmen oder hinzufügbarer interner oder externer Hardware. Hierdurch wird es ermöglicht, eine alternde Basisplattform durch Erweiterungen aktuell zu halten. Dem Kunden wird es zudem durch persönliche Modifikationen erlaubt, sein System individuell zu gestalten. Auch die Entwicklung der letzten Jahre im Mobiltelefonbereich hat gezeigt, dass Kunden individuelle Erweiterungsmöglichkeiten fordern, welche hier in Form des Verkaufs von Anwendungen über online Applikationsmärkte umgesetzt sind.

Im Automobilbereich ist eine solche Erweiterbarkeit der Fahrzeugfunktionalität ebenfalls wünschenswert. Derzeit ist dieses jedoch nach dem Verkauf kaum möglich. Beim Kauf eines Neuwagens muss sich der Käufer schon während der Bestellung über alle Wünsche bzgl. der erweiterten Ausstattungsoptionen im Klaren sein. Eine nachträgliche Erweiterung ist derzeit i.d.R. aus technischen Gründen kaum möglich. Besonders gilt dies für Fahrzeugfunktionen, die zum Kaufzeitpunkt noch gar nicht verfügbar waren.

Auch aus OEM-Sicht hätte eine Erweiterbarkeit der Fahrzeugfunktionalität enorme Vorteile. So lässt sich der Übergang von einem Fahrzeugmodell zu dessen Nachfolger fließender gestaltet, da eine Erweiterung der Fahrzeugfunktionalität durch das PnP vereinfacht wird. Hierdurch werden auch kürzere Modellintervalle möglich. Auch die Wiederverwendbarkeit bzw. die Aktualisierbarkeit der bereits vorhandenen Software wird erhöht. Gleiches gilt für den Übergang zwischen verschiedenen Fahrzeugmodellen, die auf einem ähnlichen Zentralrechner basieren.

Durch das in diesem Artikel beschriebene PnP-Konzept wird ein Weg aufgezeigt, die nachträgliche Erweiterbarkeit der Fahrzeugfunktionalität zu ermöglichen, sowohl um neue Software- als auch um neue Hardwarebausteine (z. B. Sensoren und Aktoren).

Um das PnP ohne weitere Benutzerinteraktion autonom durchzuführen, ist eine Schlüsselvoraussetzung die Möglichkeit der dynamischen Selbst-Konfiguration des Fahrzeugs. Das Vorgehen hierbei muss Safety-Anforderungen erfüllen (siehe auch [Ar12] und [Ar09]) und zudem durch geeignete Security-Mechanismen abgesichert sein (siehe Kapitel 4.3). Um die Erweiterbarkeit zu gewährleisten sind zudem standardisierte Schnittstellen notwendig.

Während des PnP-Vorgangs muss abgesichert werden, dass eine neue Funktion die vorhandenen Funktionen nicht durch Seiteneffekte beeinflusst. Dieses betrifft unter anderem das Scheduling, die Kommunikation, sowie den Zugriff auf Aktoren. Ein Mittel ist hierbei die Aufteilung unterschiedlicher Arten von Funktionen in voneinander entkoppelte Partitionen. Die Partitionierung kann anhand verschiedener Kriterien erfolgen, z. B. anhand der Sicherheitsstufen. Geeignete Datenfusions- und Arbitrierungskonzepte sind ebenfalls zur Absicherung der Seiteneffekte notwendig.

In unserem Konzept findet das PnP in der Regel in einem speziellen Wartungsmodus statt, teilweise aber auch im normalen Fahrzeugbetrieb (in einem reduzierten Umfang, welcher im Folgenden näher beschrieben wird).

PnP im normalen Fahrbetrieb:
Aus Safety-Gründen ist das PnP im Fahrbetrieb nur sehr eingeschränkt möglich. Beim Hinzufügen von neuen Softwarefunktionen in den Zentralrechner werden lediglich freie Ressourcen zur Verfügung gestellt, die zuvor für das PnP vorreserviert wurden (inkrementelles Addon). Ressourcen von bereits vorhandenen Funktionen bleiben hiervon unangetastet.

Falls für eine solche additive Konfigurationserweiterung nicht genügend freie Ressourcen zur Verfügung stehen, da beispielsweise der Verschnitt freier Ressourcen über die Kontrollrechner hinweg ungünstig ist obwohl die Gesamtressourcen ausreichen würden, wird ein Übergang in den Wartungsmodus des Systems notwendig (siehe auch „PnP im Wartungsmodus“). Hierbei kann eine Rekonfiguration stattfinden, bei der auch bereits bestehende Teile der Konfiguration verändert werden können. Währenddessen ist das Fahrzeug nicht fahrbereit.

PnP im Wartungsmodus:
Im Wartungsmodus ist es möglich, bestehende Teile der Systemkonfiguration anzupassen. So kann hier unter anderem das Deployment von Funktionen auf die Kontrollrechner geändert werden, wie auch die Vergabe von Speicher- und Rechenressourcen sowie die Netzwerkkonfiguration. Dieses erlaubt die Optimierung der Systemkonfiguration. Die Validierung der berechneten Konfiguration anhand verschiedener Kriterien ist ebenfalls Bestandteil des PnP-Vorgangs.

Dieses ermöglicht nun partiell die Integration von Funktionen, die während des Normalbetriebs nicht in die Konfiguration eingebunden werden konnten. Solche Funktionen werden bei der Initialisierung im Wartungsmodus automatisch erkannt. Ist dennoch das Hinzufügen einer neuen Funktion nicht möglich, so wird diese abgelehnt oder eine vorhandene Funktion wird deinstalliert.

Im Wartungsmodus kann das System zudem um neue Hardwaremodule ergänzt werden (z.B. Sensoren und Aktoren), die automatisch erkannt und eingebunden werden. Bei der Initialisierung wird zunächst überprüft, ob Hardwaremodule hinzugefügt oder entfernt wurden. Nachdem alle Hardwaremodule erkannt, auf Konformität geprüft und konfiguriert wurden, werden die Softwarefunktionen initialisiert. Manche Funktionen und Hardwaremodule sind obligatorisch (beispielsweise zur Umsetzung von Grundfunktionen wie Lenken, Bremsen und Beschleunigen), ohne die das Fahrzeug nicht als fahrbereit klassifiziert wird. Andere Funktionen sind hingegen optional (beispielsweise eine Einparkhilfe).

Ebenfalls ist beim PnP zu berücksichtigen, dass sich das Fahrzeug in verschiedenen Betriebsmodi befinden kann. Eine Fahrzeugfunktion kann hierbei so konfiguriert werden, dass sie nur in bestimmten Fahrzeugmodi aktiv sein soll. Bezüglich der Selbst-Konfiguration des Fahrzeuges während des PnP bedeutet dies, dass eine Konfigurationsinstanz für jeden Fahrzeugmodus berechnet werden muss.

4.2. Redundanz und Safety

Die SW-basierte Integration von Funktionen oberhalb der datenorientierten Middleware erfordert weitere Maßnahmen, wenn es sich um sicherheitskritische Funktionen handelt. Die in Kapitel 2 beschriebene Redundanz der zentralen DCCs kann zur Fehlererkennung und Maskierung genutzt werden, aber die SW-Funktionen müssen im Normalfall auf diese Redundanz angepasst werden. Ein Ziel der Middleware ist es, die Applikations-entwicklung von den Sicherheitsmechanismen der DCC-Hardware-Überwachung, -Umschaltung und Degradation zu trennen, damit die SW-Entwicklung sich auf die Funktionalität und nicht auf die Eigenheiten der DCC konzentrieren kann (horizontale Schichtung).

Die „generische“, funktionsunabhängige Safety bietet im Wesentlichen Fehlerursachenbereich eindeutig zu erkennen und das Eindringen von Fehlern darin zu verhindern [Ar09]. Ein DCC bildet als Rechnerpaar einen solchen Bereich. Beide enthaltene Rechner empfangen und senden im fehlerfreien Fall die gleichen Daten für ASIL-eingestufte Funktionen. Das sog. IO-Management prüft eingehende Daten auf Konsistenz, bevor sie den Applikationen über die MW weitergeleitet werden. Gleiches gilt für ausgehende Daten. Fehler werden nach außen gemeldet bzw. sind eindeutig von außen erkennbar, da auch die Ereignisse doppelt gesendet werden. Dies sichert die Datenkonsistenz (es werden keine fehlerhaften Daten verarbeitet).

Im Falle von Inkonsistenzen wird dieser Fehler dem *Plattform-Management* gemeldet. Dies ist eine auf allen DCCs aktive SW-Komponente, die Aufbau und Konfiguration des Systems sowie mögliche Degradationsstufen kennt. Fehlerindikatoren aus den Applikationen sowie o.g. Integritätsfehler werden hier behandelt. Reaktions-möglichkeiten umfassen: DCC-Umschaltung auf Slaves und die Abschaltung einzelner Applikationen. Das Plattform-Management sichert dadurch Aktionskonsistenz (die Zentralrechner führen die gleichen Applikationen aus).

Der zyklische Ablauf der Konsistenzsicherung (10ms Zyklen) erlaubt Fehlererkennung und Umschaltung auf einen bereits aktiven (hot-standby) Slave-DCC innerhalb einer Toleranzzeit von 50ms. Auf diesem Wege stellt die Middleware eine fehlersichere Ablaufumgebung für Applikationen zur Verfügung, die auch ASIL-D fail-operational Anforderungen genügt. Diese Sicherung der DCCs enthebt natürlich die Applikationsentwicklung nicht von dem Zwang, nach den Anforderungen der jeweiligen Kritikalitätseinstufung (ASIL-Level) zu entwickeln.

4.3. Security

Security spielt in Kraftfahrzeugen bereits heute eine wesentliche Rolle, um absichtliche Angriffe abzuwehren. Wesentliche Security-Funktionalitäten betreffen die Wegfahrsperre und Fahrzeugschlüssel, korrekter Kilometerstand, zugangsgeschützter Wartungs- und Diagnosezugang einschließlich Feature-Freischaltung und Software-Update („Flashen“) ebenso wie ein Schutz der Fahrzeug-externen Kommunikation mit Backend-Diensten, mit nomadischen Geräten und mit anderen Fahrzeugen (Car-2-X-Kommunikation). Elektrofahrzeuge werden außerdem kryptographische Security-Features für einen Manipulationsschutz des Ladeprotokolls sowie für die Unterstützung von Plug-and-Charge realisieren [Fa11]. Diese Security-Features sind teilweise in Hardware realisiert, um die Performance bzw. die Robustheit der Implementierung zu erhöhen.
Auch in der neuen Systemarchitektur müssen diese bekannten Security-Funktionen robust realisierbar sein. Der in diesem Artikel beschrieben Ansatz für die Fahrzeug-Systemarchitektur erfordert jedoch darüber hinausgehende, spezifische Security-Funktionalitäten. Die wesentlichen Treiber für Security sind dabei das Zusammenführen unterschiedlicher Arten von Funktionen auf einem Zentralrechner sowie die Erweiterbarkeit durch PnP um neue Software-Funktionen wie auch um neue Hardware-Komponenten (Sensoren, Aktoren).

1. *Zusammenführen von unterschiedlich kritischen Funktionen:* Es werden unterschiedlich kritische Funktionen in einer gemeinsamen, verteilten Ausführungsumgebung realisiert. Die Applikationen sind bezüglich ihrer Safety-Anforderung, ihrer Echtzeit-Anforderungen sowie ihrer eigenen Security-Anforderungen unterschiedlich kritisch. Diese Funktionen müssen voneinander logisch separiert werden, um eine gegenseitige Beeinflussung auch bei Ausführung in einer gemeinsamen Ausführungsumgebung zu verhindern.
2. *Erweiterbarkeit (PnP):* Es muss ermittelt werden, ob eine gewünschte PnP-Operation zulässig ist (Freigabe entsprechend Fahrzeug-Security-Policy, Lizensierung/Know-How-Schutz, für PnP-Operation berechtigter Nutzer), sodass nach einer PnP-Operation eine zulässige Konfiguration vorliegt. Bei einer PnP-Operation müssen die Mechanismen zur Separierung abhängig von den ermittelten Attributen konfiguriert werden. Mit dem Fahrzeugbus direkt dürfen nur vertrauenswürdige Komponenten direkt verbunden werden. Andere Komponenten dürfen nur eingeschränkt Zugriff auf den Fahrzeugbus erhalten.

Weitere Anforderungen an das Design der Security-Lösung kommen aus dem Realisierungs- und Anwendungsumfeld. Aufgrund der erläuterten Safety-Mechanismen

muss die Security-Lösung eine redundante Berechnung von kryptographischen Operationen, eine redundante Übertragung und ein Failover zwischen redundanten Rechnern unterstützen, ohne dass dadurch die Security selbst gefährdet wird. Außerdem ist im Fahrbetrieb ein „fail-operational“ Verhalten erforderlich, damit bei detektierten Security-Vorfällen ein einfaches Abweisen nicht immer eine angemessene Reaktion darstellt.

4.3.1. Security-Lösungsbausteine

Dieser Abschnitt gibt einen Überblick über erforderliche, für diese Architektur spezifische Security-Lösungsbausteine:

- Separierung von Funktionsdomänen: Zwischen unterschiedlichen Funktionsdomäne (z.B. Safety, Body, Komfort, Vehicle Security, Infotainment) wird durch eine Partitionierung die Rückwirkungsfreiheit erreicht. Auch bei einer Fehlfunktion oder einer erfolgreichen Manipulation eines offeneren Funktionsbereichs ist dadurch sichergestellt, dass die kritische Funktionalität nicht beeinträchtigt wird.
- Separierung von Applikationen: Einzelne Applikationen werden voneinander isoliert, sodass eine Beeinflussung durch eine andere Applikation nur über vorgesehene, zugelassene Datenpfade möglich ist, d.h. über berechtigte Zugriffe auf die Signaldatenbank. Die Applikationen sind außerdem von der Ausführungsumgebung RTE isoliert, sodass eine Applikation nicht die interne Funktion der RTE manipulieren kann. Zugriffe auf Sensoren/Aktoren sind nur indirekt über die Signaldatenbank möglich.
- Zugriffskontrolle auf Signaldatenbank: Eine Applikation kann auf Einträge der Signaldatenbank nur zugreifen, soweit sie dazu berechtigt ist.
- Eine Komponente kann nur auf Einträge der Signaldatenbank zugreifen, soweit sie dazu berechtigt ist. Nur vertrauenswürdige Komponenten dürfen direkt mit dem Fahrzeugbus verbunden sein. Andere Komponenten erhalten Zugriff über eine Anschaltkomponente „Vehicle Security Network Guard“. Diese stellt sicher, dass nur zulässige Zugriffe auf den Fahrzeugbus und damit die Signaldatenbank möglich sind.
- Authentisierung einer Software-Applikation. Eine Software-Applikation wird authentisiert, um deren für die PnP-Einbindung verwendeten Attribute korrekt zu ermitteln. Ebenso erfolgt eine Device Authentisierung von Hardware-Komponente, um deren für die PnP-Einbindung verwendeten Attribute korrekt zu ermitteln.
- PnP Security Policy: Bei einem PnP-Vorgang erfolgt eine Prüfung, ob eine PnP Operation zulässig ist. Dabei müssen unterschiedliche Kriterien und Policies unterstützt werden: Die betroffene Fahrzeugfunktionalität (Funktionsdomäne), Lizensierung, der durchführende Nutzer.
- PnP Security Konfiguration: Die vorgesehenen Security-Mechanismen zur Separierung und Zugriffskontrolle müssen bei einer PnP-Operation konfiguriert werden.

Eine robuste Implementierung dieser Security-Features in der RTE kann durch weitere Security-Maßnahmen unterstützt werden. Wesentliche Aspekte hierbei betreffen die

Schlüsselspeicherung, z.B. in einem Hardware Secure Element, eine robuste Implementierung von kryptographischen Algorithmen (Seitenkanalresistenz, Fehlerresistenz), ein Security-Hypervisor, ein sicheres Booten, Kommunikationssicherheit auf dem Fahrzeugbus und die Detektion von Manipulationen. Im Fehlerfall sind durch entsprechende Log-Einträge im Fehlerspeicher die Ursache und die Verantwortlichkeit ermittelbar.

4.3.2. Realisierungskonzept Systemarchitektur

Die RACE-spezifischen Security-Funktionen werden durch Security-Module des Runtime Environment realisiert. Für die Erweiterbarkeit (PnP-Manager) benötigte Security-Module sind:

- Device Authentication Module: Authentisierung einer Hardware-Komponente
- Application Authentication Module: Authentisierung einer Software-Applikation
- PnP Security-Policy Manager: Prüfen einer PnP-Operation auf Zulässigkeit
- Access Policy Provider: Berechtigungen bei einem PnP-Vorgang automatisch erzeugen.
- Access Policy Configurator: Berechtigungen auf Enforcement-Komponente konfigurieren.

Für die Durchsetzung der Separierung zur Laufzeit werden folgende Module verwendet:

- Signaldatenbank Access Control: Zugriffskontrolle auf Einträge der Signaldatenbank
- Vehicle Network Access Guard (Real-Time Firewall): Zugriffskontrolle auf den Fahrzeugbus und damit die Signaldatenbank durch eine Hardware-Komponente
- Hypervisor: Angriffs-resistente Separierung von Funktionsbereichen (Partitionierung)

Diese RACE-spezifischen Security-Funktionen verwenden als Basis allgemein bekannte Security-Funktionen zur Schlüsselspeicherung, kryptographische Algorithmen, sicheren Kommunikation, Policy-Prüfungen, Fahrerauthentisierung etc. Weiterhin stellt die RTE einzelnen Applikationen diese Security-Basisfunktionen bereit, um Applikations-spezifische Security-Funktionalität effizient realisieren zu können.

Innerhalb des geschlossenen RTE-Security-Kerns ist ein IT-Security-Schutz nicht zwingend erforderlich. Dabei wird jedoch ein ausreichender physikalischer Schutz der betroffenen Komponenten vorausgesetzt. Die oben beschriebenen Security-Mechanismen schützen vor möglicherweise manipulierten Komponenten, die sich außerhalb des RTE-Security-Kerns befinden, d.h. die Applikationen und die Hardware-Komponenten. Weitere Security-Funktionen können jedoch einen Beitrag leisten, den RACE-Security-Kern selbst in Ergänzung zu physikalischen Schutzmaßnahmen angriffsresistent zu realisieren.

4.3.3. Realisierungskonzept Middleware

Die zu Beginn des Kapitels 4.3 formulierten prinzipiellen Ziele der RACE-Security Architektur werden durch den Security Manager und dessen Teilmodule implementiert. Im Folgenden werden einige Module des Security Managers vorgestellt, wie sie in der Abbildung 3 dargestellt werden. Die zur Verfügung gestellten Dienste werden näher betrachtet, die erzeugten Daten und innerer Abläufe erklärt.

Zur Authentifizierung bietet der Security Manager ein *Authentification Module* an. Dieses Modul realisiert Authentifizierungsprozesse bei der Installation und der Aktualisierung von Komponenten. Diese Authentifizierung basiert zum Teil auf dem Austausch und der Überprüfung von Zertifikaten, die durch autorisierte Stellen für die Anwendung ausgestellt wurden. Daneben wird eine Authentifizierung beim Austausch von Daten zwischen verteilten Middlewareinstanzen oder Komponenten durchgeführt. Das Modul benötigt für die Durchführung dieses Dienstes einen kryptographischen Dienst.

Das *Crypto-Service Modul* bietet dem System kryptographische Dienste an. Die für die Berechnungen nötigen Schlüssel werden nicht selbst verwaltet, sondern sind zuvor von einem dafür spezifischen Dienst, dem Key-Management Modul angefordert. Beide Module bilden mit dem Secure Storage eine gesonderte vertrauenswürdige Zone, die es erlaubt direkt auf das verwaltete Schlüsselmaterial zuzugreifen und entsprechend dem erlaubten Zweck zu verwenden. Daneben verwaltet das Modul die Konfigurationseinträge für die im System vorhandenen Verschlüsselungsalgorithmen, wie z.B. symmetrische oder asymmetrische Verfahren. Dies ermöglicht ein späteres Einspielen, Aktualisieren oder Löschen von Algorithmen. Werden neue Algorithmen mittels PnP in das System eingepflegt, alte oder schwache Algorithmen entfernt oder auf den neuesten Stand gebracht, führt dies zu einer Anpassung der im Modul verwalteten und im Secure Storage Modul hinterlegten Konfiguration.

Die Verwaltung von kryptographischen Schlüsseln übernimmt das *Key Management Module*. Hier können Schlüssel in das System eingepflegt, entfernt und nach Bedarf angefordert werden. Da direkter Zugriff auf Schlüssel ein Sicherheitsrisiko darstellt, werden die Schlüssel nicht direkt verfügbar gemacht, sondern nur indirekt auf Anforderung bereitgestellt.

Das Speichern von Konfigurationsdateien, Schlüsselmaterial, Zertifikaten und weiteren Middleware immanenten Daten wird durch das *Secure Storage Module* übernommen. Dieser ist selbst verschlüsselt und bietet die Daten nur vorgelagerten Modulen nach Autorisierung über die Schnittstellen des Security Manager Moduls an. Dabei kann das Modul von in Hardware existierender Lösungen (z.B. TPM / HSM) unterstützt werden. Aber das ganze kann auch als reine Softwarelösung angesehen werden, wenn es keine Möglichkeit gibt, spezifische Hardwareunterstützung zu nutzen.

Neben den bereits vorgestellten Modulen existieren noch weitere Module, die ihre Dienste der Middleware und den Anwendungen zur Verfügung stellen. Grundsätzlich umfasst das die Dienste *Secure Communication Module* zur Bereitstellung sicherer

Kommunikation mit anderen Middlewareinstanzen und Anwendungen, sowie das *Policy Module* zum Prüfen von auszuführenden Aktionen auf deren Zulässigkeit.

Neben den sicheren Verbindungen zwischen den verteilten Middlewareinstanzen baut das *Secure Communication Module* auch die sicheren Verbindungen zwischen einzelnen verteilten Anwendungen auf. Dabei dient dieses Modul als zentrale Anlaufstelle für übergreifende Kommunikation und führt die hierfür nötigen Prozeduren und Modul-Interaktionen transparent durch. Dies schließt die Überprüfung der Verbindungs-Policies, die Authentifizierung und das konkrete Absichern der Verbindung ein.

Das *Policy Module* dient als zentrale Verwaltungs- und Entscheidungsstelle für die Anwendung von Policies. Module oder auch Anwendungen können hier Entscheidungen über geplante Aktionen oder Parameter für die Aktionen anfragen. Das Enforcement der Policy-Entscheidungen übernimmt hierbei die jeweils anfragende Instanz.

Die Überwachung der korrekten Funktionsweise wird durch den *Intrusion Monitor* realisiert. Dabei wird das Verhalten der Software zentral protokolliert und auf Auffälligkeiten untersucht. Bei einem identifizierten Angriff können lokale Reaktionen eingeleitet oder entsprechende Informationen an eine weitere den Angriff betreffende Middlewareinstanz weitergeleitet werden. Neben der Überwachung von Anwendungen beobachtet der Intrusion Monitor die Kommunikation, die über den Interface Manager verschickt wird. Hierbei arbeitet der Intrusion Monitor mit klassischen Ansätzen um schädlichen Datenverkehr zu erkennen und geeignete Gegenmaßnahmen einzuleiten.

Das *Pseudonymization Module* verändert fahrzeugbezogene Daten derart, dass diese nicht mehr dem Fahrzeug und seinem Halter zugeordnet werden können. Sollen Daten mit der Außenwelt ausgetauscht werden, müssen alle Fahrzeugbeziehungen zuerst vom Modul durch ein Pseudonym ersetzt werden. Dies soll verhindern, dass Fahrzeuginformationen aussagekräftig gesammelt und zum Nutzen dritter ausgewertet werden können. Auch verwaltet dieses Modul die Zugriffsrechte auf einzelne fahrzeugbezogene und halterbezogene Daten.

4.4. Weitere Qualitätseigenschaften

Die datenzentrische Middleware unterstützt die *Übertragbarkeit* von Datenproduzenten und -konsumenten. Sie sind voneinander entkoppelt, wodurch eine nahezu beliebige Platzierung von Funktionen auf den DCCs möglich wird. Die Entkoppelung der Funktionen sowohl untereinander als auch von der Hardware ermöglicht die effiziente, plattformunabhängige Implementierung und somit eine einfache Übertragung auf ein neues Zielsystem. Die Anpassung der Verschaltung erfolgt automatisch durch die Middleware.

Die zentralisierte Datenbank unterstützt die *Testbarkeit* des Systems. Die Ein/Ausgaben der Anwendungen sind übe die Datenbank verfügbar und die Schnittstellen können sowohl für Unit- als auch für Integrationstests genutzt werden. Traces und auch Fehlerinjektionen sind über diese Komponente der Middleware leicht möglich. Des Weiteren sind die Mechanismen der Middleware selbst leichter testbar, da die

Ergebnisse von Konsistenzprüfung, Zeitmessungen, etc. als Output der Middleware über die Datenbasis abfragbar werden.

In Zukunft werden wohl nur die kleinsten ECUs ohne *Multicore-Support* auskommen können. Erfahrung aus der Programmierung großer IT-Systeme hat gezeigt, dass eine datenfluss-zentrierte Programmierung leichter über mehrere Prozessorkerne skaliert als eine Task-synchrone Programmierung. Der vorgestellte Ansatz unterstützt einen solchen Datenfluss und erlaubt zusammen mit der Separierung von Partitionen in der RTE die Kombination mit deterministischen harten Echtzeitaufgaben. Die extrem kleinen Zykluszeiten der Aktoren werden in der vorgestellten Systemarchitektur von den koordinierenden Aufgaben der Zentralrechner getrennt.

Die *Kosten* sind ein wesentlicher Treiber für den automobilen Architekturentwurf. Der vorgestellte Ansatz ist kein Weg, Siliziumfläche oder Stücklisten zu minimieren. Vielmehr zielt er auf die Erweiterbarkeit und Entwicklungsgeschwindigkeit ab, in dem Basismechanismen der Rekonfiguration und Safety in einer Referenzarchitektur so von den Anwendungen entkoppelt werden, dass eine getrennte Anwendungsentwicklung ermöglicht wird.

5. Zusammenfassung

Die vorgestellte datenzentrische Middleware unterstützt die SW-basierte Integration von Funktionen. Über die bereitgestellten Daten lassen sich Funktionen kombinieren, die von unterschiedlichen Zulieferern entwickelt werden. Automatische Rekonfigurationsverfahren erlauben das nachträgliche Einbringen von Funktionen, dies unterstützt auch verschiedenen Entwicklungszyklen für einzelne Funktionen. Die entworfene Systemarchitektur erlaubt eine Kombination mit Safety-Mechanismen, die einen fail-operational Betrieb unterstützen.

Die Umsetzung hat im Projekt RACE begonnen [Ra12]. Die Erfahrungen können zukünftig mit vorhandenen AUTOSAR-Basisdiensten kombiniert werden und ggf. als Input für eine Erweiterung des Standards dienen; dies ist aber nicht Teil des laufenden Projekts. Das Projekt untersucht ebenfalls ein Migrationsszenario, das nicht die vollen (Safety) Garantien nutzt und damit eine schrittweise Einführung in bestehende Systeme darstellt.

Literatur

[Ar09] Armbruster, M.; Zimmer, E.; Lehmann, M.; Reichel, R. et al.: "Modularisation of Safety & Control for X-By-Wire Multiapplication-Platforms," SAE Int. J. Passeng. Cars - Electron. Electr. Syst. 1(1):8-17, 2009, doi:10.4271/2008-01-0113.

[Ar12] Armbruster, M. et al.: Ethernet meets Safety, 4. Elektronik Automotive Congress, 2012

[Br11] Broy, M.: „Mit welcher Software fährt das Auto der Zukunft?“, ATZ extra, 125 Jahre Automobil, pp. 92-97, April 2011

[Bu12] Buckl, C.; Camek, A.; Kainz, G.; Simon, C.; Mercep, L.; Staehle, H.; Knoll, A.: , "The software car: Building ICT architectures for future electric vehicles," IEEE International Electric Vehicle Conference (IEVC), 2012, pp.1-8, 4-8 March 2012

[CH12] CHROMOSOME Middleware; http://chromosome.fortiss.org

[Fa11] Falk, R.; Fries, S.: Securing the Electric Vehicle Charging Infrastructure – Current status and potential next steps, 27. vdi/VW Gemeinschaftstagung “Automotive Security”, Oct 2011, Berlin, VDI-Berichte 2131, VDI-Verlag Düsseldorf.

[Fo12] Mehr Software (im) Wagen, Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilität der Zukunft; http://www.fortiss.org/ikt2030

[HA11] Projekt HAVE-IT, FP7, Highly automated vehicles for intelligent transport; http://www.haveit-eu.org

[He01] Henzinger, T. A.; Horowitz, B.; Kirsch, C. M.: Giotto: A time-triggered language for em-bedded programming, Proceedings of the First International Workshop on Embedded Software (EMSOFT), 2001, 166 - 184.

[ISO11] ISO/DIS 26262: - Road vehicles – Functional Safety — Part 1-10

[RA12] Projekt RACE, Robust and Reliable Automotive Computing Environment for Future eCars; http://www.projekt-race.de

[Sp02] Spiegelberg, G.; „Ein Beitrag zur Erhöhung der Verkehrssicherheit und Funktionalität von Fahrzeugen unter Einbindung des Antriebstrangmoduls MOTionX-ACT®“, ISBN 9783898733670

[SP02] SPARC Secure Propulsion using Advanced Redundant Control, FP6, IST-2002-507859.

Integration von Formaler Verifikation und Test

Cyrille Comar, Johannes Kanig und Yannick Moy
AdaCore, 46 rue d'Amsterdam, F-75009 Paris (Frankreich)
{comar,kanig,moy}@adacore.com

Abstract: Üblicherweise nehmen die Verifizierungsaufgaben, die von den branchenspezifischen Standards gefordert werden, einen großen Teil der Entwicklungskosten von kritischen Systemen ein. Diese Tendenz wird noch verstärkt, da die Entwicklung und Pflege einer großen Menge von funktionalen und Robustheitstests, für immer komplexer werdende Anwendungen, nötig ist. Die formale Verifikation ist ein Weg, diese Kosten zu reduzieren, und bietet dabei gleichzeitig mehr Garantien. Neuere Standards, wie der Avionik-Standard DO-178C, erlauben die Verifizierung von Software durch formale Methoden. Im Hi-Lite Projekt streben wir die Integration von Test und formaler Verifikation für C- und Ada-Entwicklungen an. In diesem Artikel beschreiben wir die Bedingungen, unter welchen eine solche Integration die gleichen Garantien geben kann wie eine Verifikation ausschließlich durch Test. Wir beschreiben ebenfalls die Vor- und Nachteile einer solchen Methode, mithilfe einer Datenbankapplikation als Fallstudie.

1 Einleitung

In der Entwicklung von kritischer Software, wie zum Beispiel Avionik-Software, spielt der Begriff des anforderungsbasierten Tests (ABT) eine große Rolle in der Verifikation. Der DO-178B-Standard[1] [DO92] definiert verschiedene Anforderungsebenen, von denen ein Teil der Software zugeordnet wird, und die Basis für die sogenannten High-Level Requirements (HLR - Anforderungen auf hohem Abstraktionsniveau) bildet. Mithilfe dieser HLR wird dann eine Softwarearchitektur erstellt, die auch Low-Level Requirements (LLR - Anforderungen auf niedrigem Abstraktionsniveau) enthält. Diese letzteren werden oft in natürlicher Sprache formuliert, und sind bestimmten Subprogrammen[2] zugehörig. Der anforderungsbasierte Test besteht aus der Definition von Testfällen für jede Anforderung, der Überprüfung, dass diese Testfälle die entsprechende Anforderung auch abdecken, und schließlich der Implementation und Ausführung dieser Testfälle. DO-178 (6.4) definiert drei Ebenen des Tests: Hardware/Software-Integration, die auf der Hardware des Endprodukts auszuführen ist, Software-Integration, und Low-Level Tests zusammen mit Abdeckungskriterien, die

[1] Die aktuelle B-Version des DO-178-Standards wurde vor kurzem durch die C-Version ersetzt. In diesem Artikel benutzen wir das Kürzel DO-178 für beide Versionen.
[2] Wir benutzen hier Ada-Terminologie und verwenden das Wort *Subprogramm* für Prozeduren und Funktionen.

eine gute Abdeckung des Programmverhaltens garantieren. Schließlich gibt es noch Robustheitstests, die das Verhalten der Software außerhalb der spezifizierten Eingabewerte überprüfen sollen, also zum Beispiel in Fehlersituationen. Die Abdeckungskriterien, insbesondere die Modified Condition/Decision-Abdeckung (MC/DC) [Ch01], sowie die Robustheitstests, sind zu einem großen Teil für die hohen Kosten der Testaktivitäten verantwortlich. Es ist daher nur logisch, sich nach alternativen Verifikationsmethoden umzusehen, die die gleichen (oder mehr) Garantien erbringen können, und das bei niedrigeren Kosten.
Formale Methoden haben dieses Potenzial, insbesondere, weil sie all möglichen Programmverhalten analysieren können, und dadurch 100% Abdeckung erreichen. Die vertragsbasierte formale Programmverifikation, die darin besteht, dass ein Subprogramm seinen Vertrag statisch erfüllt, wurde auch schon in der Avionik-Industrie eingesetzt, für C- und SPARK-Programme [So09, HC02]. Der Anhang des DO-178C-Standards zum Einsatz von Formalen Methoden erkennt diesen Einsatz an und erlaubt explizit, bestimmte Testaktivitäten durch formale Methoden zu ersetzen. In diesem Artikel beschreiben wir, wie Low-Level Test *zum Teil* durch vertragsbasierte formale Verifikation ersetzt werden kann. Eine vollständige Ersetzung dieser Testaktivitäten würde über das Ziel hinaus schießen, da ein komplette formale Verifikation für bestimmte Teile eines Programms sehr komplex (und daher teuer) sein kann. Stattdessen streben wir eine vollautomatische formale Verifikation eines *Teils* des Programms an, während der Rest des Programms mithilfe von traditionellen Testmethoden überprüft wird. Eine Schwierigkeit besteht darin, sicherzustellen, dass diese Kombination so zuverlässig ist wie der traditionelle Ansatz, der nur auf Test beruht.

1.1 Formale ausführbare Verträge

Um formale Verifikation durchzuführen, müssen LLR in einer formalen Sprache ausgedrückt werden. In einem vertragsbasierten Ansatz werden LLR als Subprogramm-Verträge ausgedrückt. Ein solcher Vertrag besteht aus einer Vor- und einer Nachbedingung, wie es auch in der Hoare-Logik [Ho69] der Fall ist, die beide in einer geeigneten Spezifikationssprache [Ha09] formuliert sind. Die Vorbedingung beschreibt eine Eigenschaft, die ein Aufrufer herstellen muss, bevor er das Subprogramm aufrufen kann. Die Nachbedingung beschreibt eine Eigenschaft, die das Subprogramm garantiert, bevor es den Kontrollfluss wieder an den Aufrufer zurückgibt. Vertragsbasierte formale Verifikation besteht darin, separat zu überprüfen, dass jedes Subprogramm seinen Vertrag einhält, keine Laufzeitfehler (z.B. eine Division durch Null) enthält, und dass die Vorbedingungen aller aufgerufenen Subprogramme auch im Moment des Aufrufs gelten. Hier ist es nützlich, darauf hinzuweisen, dass diese Verifikation unter den folgenden Annahmen statt findet:

- Die Vorbedingung des Subprogramms ist am Anfang der Ausführung des Subprogramms wahr;
- Die Nachbedingung eines aufgerufenen Subprogramms ist nach dem Aufruf wahr.

In manchen Spezifikationssprachen, wie den SPARK-Annotationen [Ba03], werden Verträge nicht in der Programmiersprache, sondern in einer logischen Sprache ausgedrückt, während in JML [Ba03] logische und programmatische Konstrukte

gemischt werden können. Wir haben stattdessen eine vollständig ausführbare Spezifikationssprache gewählt, die völlig in die Programmiersprache eingebettet ist, wie in Design-by-Contract [Bu05, Me88]. Da Verträge von Menschen geschrieben werden, können sie auch Fehler enthalten. Durch die Wahl einer ausführbaren Spezifikationssprache können diese Verträge mit herkömmlichen Entwicklungswerkzeugen getestet und debugged werden. Insbesondere können Verträge genau wie Programme auch Laufzeitfehler enthalten, und die formale Verifikation sollte solche Fehler entdecken. So können unsere Verträge sowohl als logische Formeln, als auch als (seiteneffektfreier) Code verstanden werden. Versuche von Chalin [Ch10] haben gezeigt, dass die Akzeptanz von formalen Methoden bei Entwicklern steigt, wenn Verträge die gleiche Bedeutung wie normaler Programmcode haben.

1.2 Implizite Verträge

Der größte Beitrag zu den Kosten einer vertragsbasierten formalen Verifikation ist das Schreiben von Verträgen für Subprogramme. Wir haben eine Reihe von Entscheidungen getroffen, die diese Kosten auf ein Minimum reduzieren.

Erstens besitzt jedes Subprogramm in einer stark typisierten Sprache wie Ada schon einen interessanten Vertrag, der durch die Typen der Variablen und Parameter ausgedrückt wird. Um diese Information nutzen zu können, müssen allerdings Sprachkonstrukte, die es erlauben, das Typsystem zu umgehen, ausgeschlossen werden. Wenn dies der Fall ist, dann kann das Subprogramm als zusätzliche Vorbedingung davon ausgehen, dass alle Parameter nur Werte annehmen, die durch ihre entsprechenden Typen erlaubt werden. Genauso muss dann aber auch für die Nachbedingung gezeigt werden, dass alle Ausgabewerte in den Grenzen der entsprechenden Typen liegen.

Zweitens braucht die formale vertragsbasierte Verifikation Wissen über die Datenabhängigkeiten, also welche globalen Variablen von einem Subprogramm gelesen und/oder geschrieben werden, direkt oder indirekt. Datenabhängigkeiten werden häufig auch Seiteneffekte genannt. Es wäre eine große Bürde für den Programmierer, diese Information für jedes Subprogramm aufzuschreiben, was die Kosten für das Erstellen von Verträgen in die Höhe treiben würde. Stattdessen berechnen wir diese Information automatisch. Die Folge davon ist eine leichte Einschränkung der vorhandenen Konstruktionen für das Programm, und ein Subprogramm kann nur dann formal verifiziert werden, wenn alle direkt oder indirekt aufgerufenen Subprogramme implementiert worden sind. Gegenüber dem Test ist das natürlich keine Einschränkung.

Schließlich wird vertragsbasierte Verifikation deutlich komplexer, wenn Aliasing zwischen Objekten auftreten kann. Der manuelle Aufwand der Vertragserstellung wird größer, während der Anteil der automatischen Beweise kleiner wird. Die beste Wahl ist hier das Verbot von Zeigern in dem Teil des Programms in dem formale Verifikation durchgeführt wird, so dass kein Aliasing auftreten kann.

Auf der einen Seite sind implizite Vor- und Nachbedingungen zusätzliche Hypothesen während der Verifikation eines Subprogramms: die implizite Vorbedingung, sowie die impliziten Nachbedingungen aller Aufrufe werden als wahr vorausgesetzt. Auf der anderen Seite sind diese impliziten Vor- und Nachbedingungen zusätzliche

Verifikationsziele: die implizite Nachbedingung muss bewiesen werden und die impliziten Vorbedingungen aller Aufrufe müssen bewiesen werden.

1.3 Die Kombination von Test und formaler Verifikation

Die formale Verifikation deckt alle Verhalten eines Subprogramms ab, wogegen auch ein sehr gründlicher Test in aller Regel bestimmte Pfade des Subprogramms nicht abdeckt. Andererseits sind die Ergebnisse der formalen Verifikation auf die Annahmen gestützt, die in den Abschnitten 1.1 und 1.2 erläutert wurden, wogegen der Test keine weiteren Annahmen braucht. Wenn alle Subprogramme formal verifiziert wurden, sind diese zusätzlichen Annahmen auch berechtigt. Sie werden entweder durch Beweis, eine zusätzliche statische Analyse oder auch durch einen Metabeweis der formalen Methode selbst gestützt. Wenn ein Teil des Programms durch Test verifiziert wird, ist diese Argumentation aber nicht mehr lückenlos. Der Schlüssel zur Kombination von Test und formaler Verifikation ist daher, die Annahmen der formalen Verifikation in dem Teil zu überprüfen, der nicht formal verifiziert wird. Ob diese Kombination auf Subprogrammbasis oder auf einer gröberen Komponentenbasis erfolgt, ist dabei unerheblich, die Probleme sind die gleichen: Es werden Annahmen über einen anderen Teil des Programms gemacht, und diese Annahmen müssen gerechtfertigt werden.
Man kann sich mehrere Arten der Kombination von Test und formaler Verifikation vorstellen. Der einfachere Fall (den wir hier T+F.1 nennen) ist, die Anwendung in kleine Teile (zum Beispiel Subprogramme) aufzuteilen, und für jeden Teil nur eine der Verifikationsmethoden anzuwenden. Eine andere Möglichkeit (nennen wir sie T+F.2) ist es, jedes LLR separat mit einer der Verifikationsmethoden zu überprüfen. Wenn also ein Subprogramm mehrere LLR implementiert, sollte es möglich sein, beide Techniken für den gleichen Code anzuwenden. Schließlich (T+F.3) können bestimmte formale Eigenschaften gezeigt werden, die einen Teil oder auch die Gesamtheit der Robustheitstests ersetzen können.

1.4 Ein Fallbeispiel

Im Rahmen des Hi-Lite-Projekts [Hi10] entwickeln wir zwei Toolchains für die Kombination von Test und formaler Verifikation. AdaCore entwickelt die Toolchain für Ada-Projekte, basierend auf dem GNAT-Compiler. Das französische Institut CEA entwickelt eine Toolchain für C-Projekte, basierend auf der Frama-C Plattform und dem GCC Compiler. Beide Plattformen haben ein gemeinsames Backend für die Beweise, das von INRIA entwickelt wird, und auf der Why3-Sprache [Bo11] für Programmbeweise basiert.
Unser Tool für die vertragsbasierte formale Verifikation von Ada-Programmen ist das Tool GNATprove [GKM11]. GNATprove besteht aus einer vollständigen Kette, von Quellcode in einer sehr reichen Untermenge von Ada zu Why3 [Bo11], von dem dann sogenannte Verification Conditions (VC) generiert werden, die von dem automatischen SMT-Beweiser Alt-Ergo [Bo08] bewiesen werden. Eine VC ist eine logische Formel, die bewiesen werden muss, damit eine entsprechende Eigenschaft im Ada-Programm wahr ist. Es ist noch zu bemerken, dass GNATprove keine Aussagen über die Terminierung eines Programms macht.

In diesem Artikel beschreiben wir den Ansatz, den wir mit dem Tool GNATprove gewählt haben. Wir zeigen Beispiele anhand einer Fallstudie in Ada, die aus einer kleinen Datenbank für eine Bank besteht. Die HLR für diese Anwendung sind die folgenden:

- Es sollte möglich sein, Konten zu öffnen und zu schließen. Jedes Konto ist einer Person zugeordnet, die mit einer eindeutigen Nummer (zum Beispiel SSN, ID) identifiziert wird. Es sollte möglich sein, Geld einzuzahlen und abzuheben, und den aktuellen Kontostand abzufragen.
- Es gibt eine maximale Anzahl von 200 000 gleichzeitig offenen Konten.
- Ein Konto wird in einer zur Eröffnung festgesetzten Währung geführt. Alle Einzahlungen und Abhebungen sollten in dieser Währung stattfinden.
- Ein Konto kann nicht mehr als 1 Million, unabhängig von der Währung, beinhalten.
- Die Bank leiht kein Geld. Der Kontostand sollte nie negativ sein.

Bei der Definition einer API für das Paket `Database` werden die LLR von den HLR abgeleitet. Diese API besteht aus fünf Subprogrammen, zusammen mit der Spezifikation ihrer Funktionalität:

- `Balance`: den aktuellen Kontostand eines existierenden Kontos abfragen.
- `Open`: ein neues Konto eröffnen. Die maximale Anzahl offener Konten sollte noch nicht erreicht sein. Das neue Konto wird einem Namen und einer Identifikationsnummer zugeordnet. Der anfängliche Kontostand ist Null.
- `Close`: ein existierendes Konto, das einem Namen mit einer Identifikationsnummer zugeordnet ist, schließen. Der Kontostand sollte vor der Schließung bei Null stehen.
- `Deposit`: Einen gewissen Geldbetrag auf ein Konto einzahlen. Der Geldbetrag sollte in der gleichen Währung wie das Konto sein, und nach der Einzahlung sollte der Kontostand eine Million nicht überschreiten.
- `Withdraw`: Einen Geldbetrag von einem Konto abheben. Der Geldbetrag sollte in der gleichen Währung wie das Konto sein, und den aktuellen Kontostand nicht überschreiten.

Die Datenbank ist in einem Paket `Database` implementiert, das von den Paketen `Identity`, welches die Typen für Namen und eindeutige Identifikationsnummern definiert, und `Money`, das Geldbeträge und Währungen definiert, abhängt.

GNATprove ist in einer GPL-Version frei erhältlich [GN12] und enthält auch den vollständigen Code (weniger als 1000 Zeilen Code) für diese Beispiel-Anwendung.. Für das Verständnis mancher Details der Beispiele dieses Artikels kann es helfen, diesen Code in seiner Gesamtheit verfügbar zu haben.

2 Formale Verifikation

Sowohl die formale Verifikation von Subprogrammverträgen als auch die Verifikation der Laufzeitsicherheit bestehen aus dem Beweis, dass bestimmte Assertionen auf allen Ausführungspfaden des Programms wahr sind. Dies ist nur möglich, wenn der Quellcode

in einer bestimmten Form eingeschränkt ist, und wenn Subprogrammverträge als Assertionen ausgedrückt werden können.

3 Sprachbeschränkungen

Wir betrachten hier Ada 2012 [Sc10], die nächste Version des Ada-Standards, als Eingabesprache. Sie wird bereits voll von dem GNAT-Compiler unterstützt. Ada 2012 ist weitestgehend kompatibel mit den vorherigen Sprachversionen, also kann auch bereits existierender Ada-Code verwendet werden.

Ein große Teilmenge von Ada 2012 ist für die formale Verifikation geeignet; wir nennen diese Teilmenge *Alfa* [Al11]. Die formale Verifikation wird nur für Subprogramme versucht, deren Implementation sich in der Alfa-Teilmenge befindet. Eine noch größere Teilmenge von Ada 2012 bezeichnen wir als *Alfa Friendly* [Al11]; diese Teilmenge ist nicht unbedingt für die formale Verifikation geeignet, kann aber sicher mit Alfa-Sprachkonstrukten gemischt werden. Zum Beispiel kann ein Alfa-Subprogramm ein Alfa-Friendly-Subprogramm aufrufen, und andersherum. Die Beschränkung auf Alfa-Friendly-Code garantiert, dass globale Annahmen der formalen Verifikation gerechtfertigt sind. Kommen wir nun zu den Ideen, die den Alfa-Beschränkungen zugrunde liegen.

Zuerst geht es darum, potentielle Zweideutigkeiten im Quellcode zu vermeiden, denn sonst sind die Ergebnisse der formalen Verifikation immer zweifelhaft. Insbesondere dürfen Funktionsaufrufe keine globalen Variablen modifizieren; dies hat zur Folge, dass Alfa-Ausdrücke keine globalen Variablen modifizieren, und dies schließt Compiler-abhängiges Verhalten des Codes durch Unterschiede in der Auswertungsreihenfolge aus. Diese Einschränkung, keine globalen Variablen zu modifizieren, gilt nur für *Funktionen,* aber nicht für *Prozeduren*.

Zweitens beschränkt Alfa Sprachfeatures, um automatische Beweise zu ermöglichen. So ist bekannt, dass für bestimmte Sprachfeatures Programmbeweise deutlich schwieriger oder nicht praktikabel sind. Da wir eine vollautomatische Verifikation wünschen, schließt Alfa solche problematischen Sprachfeatures aus: Exceptions, Zeiger (in Ada *access types* genannt) und Tasking. Es ist zu beachten, dass in Ada Zeiger weniger nötig sind als zum Beispiel in C. In der Zukunft, wenn die Technologie Fortschritte macht, könnten diese bisher ausgeschlossenen Features zu Alfa hinzugefügt werden.

3.1 Formale Verträge

Ada 2012 bietet eine Reihe von Features an, um Eigenschaften von Programmen auszudrücken. Sicherlich das wichtigste Feature sind die Aspekte *Pre* und *Post*, die die Vor- und Nachbedingung eines Subprogramms durch einen Booleschen Ausdruck definieren. Die Nachbedingung kann, zusätzlich zu den existierenden Variablen und Funktionen des Programms, auf den Wert eines Parameters oder einer Variablen `V` vor dem Aufruf durch die Syntax `V'Old` zugreifen, und auch das Resultat der Funktion `F` durch `F'Result` verwenden. Der Ausdruck von Verträgen wird durch neue Syntax in Ada 2012 erleichtert: konditionelle Ausdrücke, Fallunterscheidungen, universell und existenziell quantifizierte Ausdrücke sowie sogenannte Ausdrucksfunktionen. Da Alfa-Ausdrücke keine Seiteneffekte haben, und wir durch formale Verifikation auch Laufzeitfehler ausschließen, haben formale Verträge auch eine logische Interpretation.

In unserem konkreten Beispiel, um die LLR durch Verträge auszudrücken, brauchen wir zuerst Hilfsfunktionen, die den Zustand der Datenbank abfragen:

- `Existing`: gibt an, ob ein Konto existiert.
- `Belongs_To`: gibt an, ob ein Konto zu einem Namen und einer Identifikationsnummer gehört.
- `Max_Account_Reached`: gibt an, ob die maximale Zahl von Kontos erreicht wurde.
- `Currency`: gibt die aktuelle Währung eines Kontos an.

Diese einfachen Funktionen kann man am besten als Ausdrucksfunktionen definieren. Zum Beispiel wird die Funktion `Existing` wie folgt direkt durch einen Ausdruck, der selbst wieder einen Funktionsaufruf enthält, definiert:

```
  function Existing (Account : in Account_Num) return
Boolean is
     (not Is_Available (Account));
```

Die Funktion `Is_Available` beschreibt die Account-Slots, die noch verfügbar sind, das heisst, ein Konto existiert, wenn der entsprechende Slot nicht mehr verfügbar ist.
Mithilfe solcher Funktionen können alle LLR aus Abschnitt 1.4 durch Alfa-Verträge ausgedrückt werden. Wir geben hier als Beispiel die Verträge von `Balance` und `Open` an:

```
  function Balance (Account : in Account_Num) return
Money.Amount
  with
    Pre => Existing (Account);

  procedure Open
    (Customer : in     Identity.Name;
     Id       : in     Identity.Id;
     Cur      : in     Money.CUR;
     Account  :    out Account_Num)
  with
    Pre  => not Max_Account_Reached,
    Post => Existing (Account)
              and then Belongs_To (Account, Customer, Id)
              and then Money.Is_Empty (Balance (Account));
```

3.2 Verifikation von impliziten Verträgen

Wie bereits in Abschnitt 1.2 beschrieben, sind implizite Verträge für Datenabhängigkeiten automatisch korrekt, da sie auch für die Programmteile erstellt werden, die nicht in Alfa sind. Implizite Verträge, die aus starker Typisierung und Non-Aliasing hervorgehen, sollten aber explizit verifiziert werden.

Für die starke Typisierung ist es ausreichend, VCs für die entsprechenden impliziten Vorbedingungen und Nachbedingungen zu generieren, die dann wie alle anderen VCs bewiesen werden. In unserem Beispiel müssen bei einem Aufruf der Prozedur `Open` drei VCs generiert werden, die zeigen, dass die drei Parameter `Customer`, `Id` und `Cur` Werte haben, die von ihrem Typ erlaubt werden; und wir müssen eine VC generieren, die zeigt, dass der Ausgabeparameter `Account` beim Ende des Prozeduraufrufs einen Wert hat, der von seinem Typ erlaubt wird.
Übrigens nehmen wir nicht automatisch an, dass globale Variablen gültige Werte haben, es sei denn, sie werden bei der Deklaration auch gleich initialisiert. Wenn es nötig ist, muss der Programmierer die entsprechenden Vor- und Nachbedingungen hinzufügen. In dem Fall der Prozedur `Open` sind alle vier globalen Variablen direkt bei der Deklaration initialisiert.
Für Non-Aliasing gibt es eine statische Überprüfung, dass keine zwei Parameter, oder ein Parameter und eine globale Variable, Alias sind, außer:

- beide werden nur gelesen,(zum Beispiel zwei `in`-Parameter)
- einer von beiden Werten ist ein Skalar, und wird mit pass-by-value übergeben.

Im Beispiel liest und schreibt die Prozedur `Open` vier globale Variablen: den Skalar `First_Available` und die Arrays `Links`, `Accounts` und `Accounts_Balance`; und sie hat einen nur gelesenen skalaren Parameter `Cur`. Daher müssen wir für jeden Aufruf von `Open` überprüfen, dass die folgenden Paare von Argumenten und globalen Variablen nicht gealiast sind: `Customer` und `Account`, `Id` und `Account`, und jedes der drei Argumente (`Customer`, `Id`, `Account`) gepaart mit einer der globalen Variablen. Die meisten dieser Paare können nicht Alias sein, da sie unterschiedliche Typen haben und Alfa stark getypt ist. Das einzige wirklich zu überprüfende Alias-Paar ist das Argument `Account` und die globale Variable `First_Available`. Dies ist statisch durch eine einfache semantische Überprüfung zu erreichen.

3.3 Verifikation von expliziten Verträgen

Für vom Programmierer eingegebene Verträge sollten VCs generiert werden, und an einen automatischen Beweiser weitergegeben werden.
In unserem Beispiel gibt es im Fall der Funktion `Balance` keine Nachbedingung zu beweisen. Aber die Funktion `Balance` hat eine Vorbedingung, die bei jedem Aufruf bewiesen werden muss. Dies ist sowohl in Programmcode als auch in Verträgen der Fall, da ein Funktionsaufruf auch in einem Vertrag seine Vorbedingung erfüllen muss. Zum Beispiel sollte eine VC generiert werden, um zu beweisen, dass die Vorbedingung von `Balance` bei dem Aufruf in der Nachbedingung von `Open` erfüllt ist. Dies ist einfach, da die benötigte Bedingung `Existing (Account)` bereits in der Nachbedingung von `Open` gefordert wird.
Bei der Prozedur `Open` gibt es eine Nachbedingung. Also sollte eine VC erstellt werden, die ausdrückt, dass in jedem Kontext, wenn die Vorbedingung von `Open` wahr ist, die Prozedur in einem Zustand endet, in dem die Nachbedingung wahr ist.
GNATprove beweist automatisch alle Verträge in unserem Beispiel: 17 Vorbedingungen von Aufrufen und 7 Nachbedingungen von Subprogrammen.

3.4 Verifikation der Laufzeitsicherheit

Für alle möglichen Laufzeitfehler, die als Assertion ausgedrückt werden können (alle Laufzeitfehler bis auf Stack Overflow in Alfa Code), sollte eine VC erstellt und bewiesen werden.
In Ada 2012 können Verträge und Programmcode Laufzeitfehler enthalten. Also sollten VCs auch für mögliche Laufzeitfehler in Verträgen erstellt und bewiesen werden. Vorbedingungen verdienen eine spezielle Behandlung, da der Ausführungskontext in der Regel nicht bekannt ist; um Laufzeitfehler völlig auszuschließen, sollten Vorbedingungen sich selbst durch Tests vor Laufzeitfehlern schützen.
In unserem Beispiel können die Funktion `Balance` und die Prozedur `Open` Laufzeitfehler wegen Bereichsüberprüfungen verursachen, insbesondere durch Array-Zugriffe. In beiden Subprogrammen kann die Vorbedingung keinen Laufzeitfehler auslösen.
GNATprove beweist automatisch alle Laufzeitüberprüfungen in unserem Beispiel, und damit dessen Laufzeitsicherheit: 32 Indexüberpüfungen für Arrayzugriffe, 16 Bereichsüberprüfungen und 10 Überlaufsüberprüfungen.

3.5 Zusätzliche Überprüfungen

So wie Programmcode werden auch Verträge von Menschen geschrieben und können Fehler enthalten. Wir haben einige Vorschläge, wie Unstimmigkeiten in Verträgen automatisch entdeckt werden können, um Fehler so zeitig wie möglich und auch ohne eine existierende Implementation zu finden. In jedem der folgenden Punkte wird eine bestimmte VC generiert:

- Eine überflüssige Annotation ist die Wiederholung eines schon bekannten Fakts; dies kann durchaus für den Benutzer schwer zu sehen sein. Das folgende Beispiel zeigt dies:
  ```
  Pre => Existing (Account) and then not Is_Available (Account);
  ```
  Hier ist `Existing` als das Gegenteil von `Is_Available` definiert, also ist der zweite Teil der Vorbedingung unnötig.
- Eine ungültige Annotation ist eine Annotation, die nie gültig sein kann. Wie zuvor kann das manchmal schwer zu sehen sein. Die folgende Vorbedingung kann nie erfüllt werden:
  ```
  Pre => Existing (Account) and then Is_Available (Account);
  ```
- Ein unimplementierbarer Vertrag kann wie folgt definiert werden: Es gibt Eingangswerte des Subprogramms, die die Vorbedingung erfüllen, aber für welche keine Ausgabewerte existieren, die auch die Nachbedingung erfüllen. Anders ausgedrückt gibt es keine Implementation, die den Vertrag erfüllen kann. Es ist nicht immer einfach, zu sehen, dass ein Vertrag unimplementierbar ist, wie das folgende Beispiel zeigt:
  ```
  function Sign (X : Integer) return Integer with
       Post => (if X <= 0 then Sign'Result = -1)
                  and then (if X >= 0 then Sign'Result = 1) ;
  ```

Hier liegt das Problem bei dem Wert 0 für X, da es dann unmöglich wird, ein Resultat zu finden, das sowohl gleich -1 als auch gleich 1 ist.

Diese VCs für zusätzliche Überprüfungen stehen nicht im Zusammenhang mit dem Beweis von Verträgen oder der Laufzeitsicherheit. Hier sind wir nur daran interessiert, einige davon zu beweisen (oder eben nicht), aber nicht unbedingt alle. Diese VCs können redundante, unzulässige, oder unimplementierbare Verträge aufzeigen, aber wir wollen nicht beweisen, dass ein Vertrag nicht redundant, gültig, oder implementierbar ist.

4 Gemischte Verifikation

Wir möchten formale Verifikation auf so viele Programme wie möglich anwenden, inklusive Programme, die nicht zu 100% in Alfa sind (weil sie zum Beispiel Zeiger verwenden, oder Exceptions, und so weiter). Wir sind daher an einer Methode interessiert, die formale Verifikation für Subprogramme in Alfa erlaubt, und gleichzeitig ermöglicht, auf andere Subprogramme Test anzuwenden. Es ist offensichtlich, dass das Resultat einer solchen Kombination nicht so zuverlässig sein kann wie das Resultat einer kompletten formalen Verifikation. Stattdessen möchten wir garantieren, dass die Kombination so hohe Garantien wie der Test allein ergibt. In Prozessen, die heute ausschließlich auf Test beruhen, können wir dadurch bestimmte Tests durch formale Verifikation ersetzen.

4.1 Beschränkungen der Programmiersprache

Wir müssen die Sprachfeatures, die im gesamten Programm eingesetzt werden, beschränken, um die Annahmen, die die formale Verifikation trifft, nicht zu gefährden, und sie auf der Testseite überprüfen zu können. Trotzdem wollen wir Zeiger und Exceptions in diesen Programmteilen erlauben, auch wenn sie nicht in Alfa sind.
Datenabhängigkeiten sind transitiv. Um die Datenabhängigkeiten für Alfa-Programme genau berechnen zu können, müssen wir sie auch für Alfa-Friendly-Subprogramme berechnen können, die zum Beispiel von Alfa-Subprogrammen aufgerufen werden. Um die Korrektheit dieser Datenabhängigkeiten zu garantieren, unterscheiden wir zwischen Variablen, auf die gezeigt werden kann, und die, auf die nicht gezeigt werden kann. Die letzteren sind die einzigen globalen Variablen, die zu Alfa gehören. Die ersteren fassen wir in einer einzigen logischen Variable, die wir `Heap` nennen, zusammen, die alle erreichbaren Speicherbereiche darstellt. Wir verbieten auch Subprogramm-Aufrufe durch Zeiger, deren Datenabhängigkeiten nicht einfach berechnet werden können, sowie sogenannte Controlled Types, deren Semantik das Einfügen von vielen Aufrufen an verschiedenen Stellen des Aufruf-Graphs benötigt, und daher die Berechnung der Datenabhängigkeiten erschwert.
Diese Beschränkungen sind genug, um zu garantieren, dass Aliasing entdeckt werden kann, wenn es die Annahmen der formalen Beweise gefährdet.
Die obigen Beschränkungen ergeben zusammen die Alfa-Friendly Teilmenge von Ada 2012. Sie garantieren dass die impliziten Verträge für Datenabhängigkeiten, starke Typisierung und Non-Aliasing gewährleistet werden können. Formale Definitionen von Alfa und Alfa-Friendly sind Teil des erwarteten Ergebnisses des Hi-Lite-Projekts. Einige

dieser Beschränkungen können später aufgehoben werden, zum Beispiel könnten Subprogramm-Aufrufe durch Zeiger mit einer Analyse der möglichen Werte des Zeigers behandelt werden. Um es zusammenzufassen, ist in unserem Kontext eine formale Verifikation nur dann möglich, wenn das gesamte Programm Alfa-Friendly ist, und die formale Verifikation kann nur auf Alfa-Programme angewendet werden.

4.2 Verifikation von benutzerdefinierten Verträgen

Wir nehmen in diesem Abschnitt an, dass sowohl der Test als auch die formale Verifikation geeignet sind, um eine ausreichende Fehlerfreiheit eines Subprogramms zu garantieren. Ist diese Annahme akzeptiert, gibt es nur noch die Fälle an der Schnittstelle zu begutachten.
Der erste Fall ist, wenn ein getestetes Subprogramm `T` ein bewiesenes Subprogramm `P` aufruft. Die Verifikation von `P` beruht auf der Annahme, dass seine Vorbedingung bei jedem Aufruf gilt. Diese Annahme kann beim Test von `T` überprüft werden, indem die Vorbedingung von `P` ausgeführt wird.
Der zweite Fall ist, wenn ein bewiesenes Subprogramm `P` ein getestetes Subprogramm `T` aufruft. Die Verifikation von `P` basiert auf der Annahme, dass, wann immer `T` in einem Kontext aufgerufen wird, der die Vorbedingung einhält, auch die Nachbedingung eingehalten wird. Zuerst sollte also die Vorbedingung von `T` das LLR ausdrücken, so dass es tatsächlich korrekt ist, `T` unter diesen Bedingungen aufzurufen. Zweitens kann die Annahme des Beweises überprüft werden, indem die Nachbedingung von `T` ausgeführt wird.
Beide Verifikationen sind nur möglich, da Verträge ausführbar sind.

4.3 Verifikation von impliziten Verträgen

Implizite Verträge für Datenabhängigkeiten sind automatisch korrekt. Die Verifikation von Annahmen, die mit starker Typisierung sowie Non-Aliasing zu tun haben, erfordert zusätzliche Laufzeitüberprüfungen.
Implizite Verträge für starke Typisierung können durch Test überprüft werden, genau so wie für benutzerdefinierte Verträge, indem die entsprechende Assertion ausgeführt wird. In unserem Beispiel, für die Prozedur `Open`, sollten drei Assertionen am Anfang des Subprogramms ausgeführt werden, nämlich dass die Variablen `Customer`, `Id` und `Cur` Werte annehmen, die von ihrem Typ erlaubt sind, und genauso am Ende des Subprogramms, für den Ausgabeparameter `Account`.

```
procedure Open
  (Customer : in     Identity.Name;
   Id       : in     Identity.Id;
   Cur      : in     Money.CUR;
   Account  :    out Account_Num);
```

Es ist auch zu beachten, dass der Begriff, einen Wert zu haben, der von dem Typ erlaubt wird, nicht mit Initialisierung gleichzusetzen ist. Während eine initialisierte Variable immer einen vom Typ erlaubten Wert hat, so ist das Gegenteil nicht unbedingt wahr:

Eine Variable kann einen von ihrem Typ erlaubten Wert haben, ohne initialisiert worden zu sein. Insbesondere ist das immer der Fall für Integer-Typen, die den Integerbereich der Maschine darstellen.
Für Non-Aliasing zwischen Subprogramm-Parametern sollten wir überprüfen, dass die Argumentobjekte paarweise nicht überlappen, siehe auch Abschnitt 3.2. Für die `Open`-Prozedur muss überprüft werden, ob `Customer` und `Account` überlappen, oder `Id` und `Account` überlappen. Für das Non-Aliasing zwischen einem Subprogramm-Argument und einer globalen Variable ist eine globale statische Analyse nötig.
Für die Prozedur `Open` bedeutet dies, dass all Aufrufe, die die globale Variable `First_Available` als Argument übergeben, statisch als Fehler gekennzeichnet werden.
In der Praxis werden wir einen speziellen Mode des GNAT-Compilers hinzufügen, der zusätzliche Assertionen in den ausführbaren Code einfügt, um die oben beschriebenen Laufzeitüberprüfungen durchzuführen.

4.4 Formale Testfälle

Im allgemeinen werden Testfälle durch handgeschriebene Orakel erzeugt, die die LLRs eines Subprogramms zu Code übersetzen. Wenn diese Orakel als Assertionen geschrieben worden, können sie auch als Verträge geschrieben werden. Wenn dies getan wurde, wird der Test eines Subprogramms auf den Aufruf dieses Subprogramms mit geeigneten Eingabewerten reduziert, denn die Assertionen werden ja als Bestandteil des Vertrags ausgeführt.
Ein Vertrag beschreibt das allgemeine erwartete Verhalten eines Subprogramms in einem bestimmten Kontext. Wenn die Verifikation auf Test basiert, empfehlen wir, das Subprogramm mit repräsentativen Eingaben auszuführen, die den Zustandsraum abdecken. Anders ausgedrückt sollte es genug Testfälle geben, um die LLR funktional abzudecken. Da der Test nicht vollständig ist, kann es sein, dass das Subprogramm in einem unerwarteten Zustand aufgerufen wird, für den ebenfalls ein erwünschtes Ergebnis angegeben werden sollte, um den Code so robust wie möglich zu machen. DO-178 unterscheidet zwischen diesen beiden Typen von Testfällen:

- die *Normalbereich*-Testfälle, die das Subprogramm relativ zu seinen Requirements testen, und
- die *Robustheits*-Testfälle, die das Verhalten des Subprogramms in unnormalen Umständen, außerhalb der spezifizierten Bedingungen, testen.

Oft können Testfälle als zusätzliche Verträge ausgedrückt werden, die ein Subprogramm erfüllen sollte. Mit der Möglichkeit in Ada 2012, compilerabhängige Aspekte hinzuzufügen, haben wir im GNAT-Compiler den Aspekt `Test-Case` hinzugefügt, der es Benutzern erlaubt, Testfälle, die zu einem Subprogramm gehören, zusammen mit den Verträgen zu definieren. Ein Testfall ist entweder im Normalbereichs- oder im Robustheitsmodus, so wie oben beschrieben. Ein Nominaler Testfall *erweitert* die Vor- und Nachbedingungen, in dem er Requires- und Ensures-Aspekte definiert. Ein Robustheits-Testfall *ersetzt* die existierenden Vor- und Nachbedingungen mit den gleichen Aspekten.
Zum Beispiel können wir für unsere Funktion `Balance` wie folgt Testfälle definieren:

```
  function Balance (Account : in Account_Num) return
Money.Amount
  with
    Pre => Existing (Account),
    Test_Case =>
        (Name    => "existing account",
         Mode    => Nominal,
         Ensures =>
              Balance'Result.Currency       =       Currency
(Account)),
    Test_Case =>
        (Name     => "unknown account",
         Mode     => Robustness,
         Requires => not Existing (Account),
         Ensures  => Balance'Result.Currency = None);
```

Der Testfall “existing account“ wird im nominalen Modus ausgeführt, in dem auch der Vertrag erfüllt werden soll. Hier wollen wir testen, dass die Währung des Resultats die gleiche wie die des Kontos ist. Der Testfall “unknown account“ exerziert diese Funktion außerhalb ihres Vertrags; hier ignorieren wir den Vertrag von `Balance`. Stattdessen fordern wir, dass `Balance` mit einem nicht existierenden Konto aufgerufen wird, und dass das Resultat eine ungültige Währung besitzt.
Wir haben ein Tool entwickelt, GNATtest, das Unit-Tests basierend auf Testfällen erlaubt. Es unterstützt Ada 2012 Subprogramme, die mit Testfällen und Verträgen annotiert sind. GNATtest erstellt ein Testgerüst für so ein Projekt, mit einer separaten Testprozedur für jeden Testfall. Der Benutzer muss dann jede Testprozedur mit einem korrekten Aufruf des zu testenden Subprogramms füllen, und kann dann das Testgerüst kompilieren und ausführen um einen ausführlichen Testbericht zu erhalten.
Es ist zu beachten, dass der `Test-Case`-Aspekt nicht zur Beschreibung von *konkreten Tests* gedacht ist, etwa “mit dem Parameter `X` gleich 2 soll das Subprogramm 4 zurückgeben“, sondern für *Testfälle*, also etwa “wenn der Parameter `X` positiv ist, sollte das Resultat auch positiv sein“.
Die Definition von formalen Testfällen erlaubt es, eine automatische Verifikation durchzuführen, ob die Testfälle auch den Vertrag abdecken.

5 Zusammenfassung

Fortschritte im Stand der Technik der automatischen Beweiser machen es möglich, aussagekräftige statische Garantien für Programme auf kosteneffiziente Art und Weise zu erlangen. Um solche formalen Verifikationsmethoden in den Verifizierungsprozess einzubauen, müssen die Ergebnisse dieser Methoden mit denen von herkömmlichen Unit-Tests kombiniert werden. Das Hi-Lite-Projekt erreicht dies, wie im Verlauf dieses Dokuments anhand einer Datenbank-Anwendung gezeigt wurde. Die größten Kosten verursacht dabei die Formulierung der Verträge für jedes Subprogramm; es ist aber bekannt, dass diese schon an sich einen Nutzen haben: formale Spezifikationen führen zu weniger Fehlern, sogar wenn gar keine Analyse angewendet wird. Wird dies aber getan, gibt es natürlich noch weitere Vorteile: Im Testbetrieb dienen Verträge als

zusätzliche Assertionen, und man kann schneller sehen, wo der Fehler liegt. Beweise können mithilfe von Verträgen die Laufzeitsicherheit sowie die funktionale Korrektheit des Programms beweisen. Und schließlich können zusätzliche Überprüfungen die Stimmigkeit der Verträge selbst analysieren.
Wir zeigen in diesem Dokument die Kombination der Resultate für Subprogramme, die entweder insgesamt formal bewiesen, oder gründlich getestet wurden. Diese Kombination haben wir eingangs T+F.1genannt. Wir haben gezeigt, dass diese Kombination so hohe Garantien wie der Test allein erreichen kann, wenn das vollständige Programm bestimmte Einschränkungen einhält und zusätzliche Laufzeitüberprüfungen während des Tests aktiviert werden. Industrielle Fallstudien, die auf den hier vorgestellten Werkzeugen basieren, haben bereits begonnen, Ergebnisse werden in 2013 erwartet.
Wir glauben, dass auch Interesse an einer feineren Kombination von Test und formaler Verifikation besteht. So könnte beispielsweise bewiesen werden, dass die vorhandenen Testfälle für ein Subprogramm vollständig die Vorbedingung abdecken. Als ein Beispiel, schauen wir auf die beiden Testfälle des `Open`-Subprogramms aus unserem Datenbank-Beispiel:

```
Test_Case => (...
              Requires => Num_Accounts = 0,
              ...),
Test_Case => (...
              Requires => Num_Accounts > 0,
              ...),
```

Hier sehen wir deutlich, dass die beiden Testfälle den erlaubten Zustandsraum von `Open` in zwei disjunkte Räume teilen, und ihn genau abdecken, da `Num_Accounts` als natürliche Zahl definiert ist. Dies kann auch durch eine logische Formel ausgedrückt werden: Die `Requires` aller Testfälle, mit einem `or` verbunden, sollten äquivalent zu der Vorbedingung sein.
Zweitens können Testfälle auch dazu verwendet werden, als Schnittstelle zwischen formaler Verifikation und Test zu dienen. Zum Beispiel könnte, wenn der Beweis des regulären Vertrags nicht erbracht werden kann, wenigstens versucht werden, einen oder mehrere Testfälle zu beweisen. Diese Kombination von Test und formaler Verifikation auf der Basis des Testfalls, die wir in der Einleitung T+F.2 genannt haben, würde aber eine andere Art des Beweises der Vollständigkeit erfordern. Der Begriff der Vollständigkeit, wie er zum Beispiel von der Testabdeckung geliefert wird, ist hier nicht anwendbar.
Drittens können Verträge in vielen Fällen vervollständigt werden, um das erwartete Verhalten in unnormalen Situationen zu beschreiben. Indem man diese Verträge beweist, kann man das Ziel des Robustheitstests erreichen, dass wir in der Einleitung T+F.3 genannt haben. Eine andere Art, mit Robustheit mithilfe formaler Methoden umzugehen, wäre, zu beweisen, dass die Vorbedingung von allen aufrufenden Subprogrammen erfüllt wird. Es ist offen, ob diese selbst bewiesen werden müssten, oder ob ein Teil dieser Beweise ausreicht.

Literatur

[Al11] Alfa RM, 2011. http://www.open-do.org/wp-content/uploads/2011/12/alfa_rm.pdf

[Ba03] J. Barnes. High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Bo08] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In SMT'08, volume 367 of *ACM ICPS*, pages 1–5, 2008.

[Bo11] F. Bobot, J.-C. Filliâtre, A. Paskevich, and C. Marché. Why3: Shepherd your herd of provers. In Proceedings of Boogie 2011, the 1st International Workshop on Intermediate Language Verification, Aug. 2011.

[Bu05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf., 7:212–232, June 2005.

[Ch10] P. Chalin. Engineering a sound assertion semantics for the verifying compiler. IEEE Trans. Softw. Eng., 36:275–287, March 2010.

[Ch01] J. J. Chilenski. An Investigation of Three Forms of the Modified Condition/Decision Coverage (MCDC) Criterion. Technical Report DOT/FAA/AR-01/18, Apr. 2001.

[DO92] RTCA. Software considerations in airborne systems and equipment certification. Document RTCA DO-178B, 1992.

[GKM11] J. Guitton, J. Kanig, and Y. Moy. Why Hi-Lite Ada? In Proceedings of Boogie 2011, the 1st International Workshop on Intermediate Language Verification, Aug. 2011.

[GN12] http://www.open-do.org/projects/hi-lite/gnatprove/

[Ha09] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. technical report CS-TR-09-01a, University of Central Florida, School of EECS, 2009.

[HC02] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. IEEE Software, 19(1):18–25, 2002.

[Hi10] Hi-Lite: Simplifying the use of formal methods. http://www.open-do.org/projects/hi-lite/

[Ho69] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12:576–580, October 1969.

[Me88] B. Meyer. Object-Oriented Software Construction. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[Sc10] E. Schonberg. Towards Ada 2012: an interim report. In Proceedings of the ACM SIGAda annual international conference on SIGAda, SIGAda '10, pages 63–70, New York, NY, USA, 2010. ACM.

[So09] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In Proceedings of the 2nd World Congress on Formal Methods, FM '09, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.

Constructive Requirements Modeling – More Reliable Implementations in a Shorter Time

Christian Berger[1], Sebastian Siegl[2]

[1]Chalmers | University of Gothenburg, Sweden
[2]Audi Electronics Venture GmbH, Ingolstadt, Germany

christian.berger@chalmers.se
sebastian.siegl@audi.de

Abstract: Requirements engineering is nowadays the broadly accepted method to manage customer's requirements. The result is a specification from which a solution is implemented and which is used to validate the realization in terms of their fulfillment. However, today's tools assist in organizing and tracking the requirements but reliable criteria about their completeness, consistency, and realizability are missing. Furthermore, the resulting artifact is a document, which must be read and understood by humans, which itself is error-prone. It is obvious that errors and ambiguities result in an unwanted solution which is often and in the worst case only discovered in the final stage: Testing. This paper outlines an approach for constructive requirements modeling, which describes completely customer's demands in a formal manner so that already during the requirements' elicitation inconsistencies are eliminated, completeness is assessed, realizability is ensured, and all valid test cases can be derived by using a model-based testing approach. Therefore, we propose adaptions to the traditional V-model to not only save valuable development and testing time but also to achieve better results. The applicability is shown on the example of the software for an auxiliary heating system at a large German OEM.

1 Introduction and Motivation

Today's comfort and safety systems are not limited to the premium segment anymore. Instead, they are available to a cost-elastic and very competitive market. Driven by competitors, which equip their vehicles for the mass-market with more and more software-intense vehicle functions on the one hand but focused to limit the increasing development costs on the other hand, OEMs demand appropriate methods to master the overall costs while focusing on the quality of a vehicle function. According to [KBP01, Sel07], the later a manufacturer focuses on quality and correctness of a system the higher are the costs – often orders of magnitudes depending on the current development stage. Regarding the widely and often in iteration-cycles used V-model, an OEM applies

quality methods during or after the implementation stage first to determine the fulfillment of the specified requirements.

Thus, today's requirements engineering tools provide sophisticated assistance for managing a customer's requirements on the one hand [GNA+12]; furthermore, a specific requirement can be associated with one or many test cases to testify and track its fulfillment. Thus, the quality manager is always and at any time informed about the current nominal quality of the implementation.

However on the other hand, the consistency, completeness, realizability, and thus the actual quality of requirements is not assessed at all. Considering today's increasingly complex software systems, which demand a more elaborated software engineering [PBKS07] and therefore methods for quality assurance of these software-intense vehicle functions [RBK06, BBHR07, BBKR09, Ber10, BR12], the risk also increases to oversee possible situations with which a vehicle function could be faced and which could threat participants within the system context like pedestrians or bicyclists. However, this fact is not acceptable for safety systems in a customer product like vehicles!

To cope with these risks, OEMs and automotive suppliers are required to develop according to the ISO-26262 [ISO11], which defines obligatory processes and methods regarding the defined Automotive Safety Integrity Level (A-SIL); the ISO-26262 explicitly demands for software safety requirements that they must be "comprehensible, precise, unambiguous, verifiable, and realizable" and states furthermore that "traceability and adequateness regarding the SIL level and architecture" [p.99, ISO11] must be ensured. However, only semi-formal methods are "highly recommended" and thus, rather mandatory w.r.t. a specific A-SIL level.

In contrast, the avionics sector [DO11] recommends explicitly formal methods depending on the effect of a possible failure to extend classical testing approaches. And only in the railway sector [EN01], formal methods are highly recommended for using as a relevant technique even during the requirements specification.

Advantages of formal models are for example their mathematical proof of consistency, completeness, or realizability. Thus, the application of formal models to a requirements specification could significantly improve their quality due to these characteristics before its actual implementation is started. Furthermore, available methods from the requirements engineering management can be reused, for example traceability for single requirements or exchangeability with suppliers. However, there are still some concerns about formal methods, like additional costs, increased time for their application, and the lack of adequate tooling.

In this contribution, the method of sequence-based specification (SBS) [PP03, Car09] is described as a formal method to support the process of requirements analysis and formalization to achieve the aforementioned advantages. Therefore, we propose to create an additional artifact besides the traditional requirements document during the first stage in the development, which is verified and serves all later stages of the development process. Furthermore, we suggest to rethink the traditional V-model based development process to explicitly rely on this additionally created artifact to shorten development and

testing time on the one hand; on the other hand, the aforementioned characteristics of formal models yield to more reliable results.

The paper is structured as follows: First, the applicability of formal methods during the development of automotive software-intense systems is discussed. Next, theoretical principles of the SBS are presented. Afterwards, a revised V-model based development process that explicitly embraces the resulting artifacts from the SBS is proposed, which addresses the mentioned caveats. The process itself was successfully applied during the development of an auxiliary heating system, which is briefly outlined in the end.

2 Related Work

Applying formal methods during the requirements specification of software systems is known for more than two decades. Already in [GCR94], the application of formal methods is described for four different systems: First, the use of formal methods to specify a shutdown system for a nuclear plant; next, their usage for verification and validation of a safety-critical signaling system for the subway system of Paris to reduce the time gap between two trains from 150s to 120s; then, formal methods in terms of graphical notations to specify subsystems of a traffic alert and collision avoidance system for planes; and finally, formal methods were required within a project from the US Department of Defense for security-critical networks.

Thus, already at that time, the importance of formal methods was recognized to achieve the common goal: To increase the customer's confidence in the resulting implementations for these different, complex, and safety-critical systems. Nothing less than the same goal from the customer's perspective applies for modern safety-critical systems that are built into a mass-product like vehicles! However yet today, the aforementioned concerns about formal methods were basically confirmed by [Kon11] who carried out interviews to gather information about software testing and practical application of formal methods in automotive projects.

Although the ISO-26262 is in effect since November 2011, discussions about adequate formal methods, which comply with this standard, are still insufficient. One reason could be their limited recommendation within the standard as already mentioned. Moreover, recent publications like [CJL+08] focus only on annotation support for requirements as well as externally carried out verifications on top of AUTOSAR to support the requirements' traceability.

The authors of [PHP11] propose an approach to check automatically the vacuity of requirements. Compared to the SBS, it is a subsequent processing approach to ensure the validity of the resulting requirements specification, while SBS is constructing this specification at first.

Previous work to this paper which also bases on SBS was outlined in [SHG10] and [SHGB11]. While the former outlines the successful application of SBS during Hardware-in-the-Loop (HiL) testing on the example of an energy management system,

the latter deals with the successful application of SBS already during Model-in-the-Loop (MiL) testing of an automotive safety system in pre-series development project already at the early development stages.

The results of that previous work encouraged us to focus on the development process itself and to rethink the usage of formally modeled requirements already at the very earliest stage instead of their usage during MiL- or even HiL-testing only. Thus, as an extension to our previous work, this contribution focuses on revising the traditional and widely accepted V-model based development process to tackle the increasing complexity of software-intense systems by relying on constructive requirements modeling using SBS.

3 Constructive Requirements Modeling

The fundamental idea behind the SBS and its method to create an executable specification is described in the following. It is a constructive method, which means that properties like completeness of the resulting work products do not need to be verified by additional activities. They are correct w.r.t. the initially given preconditions and assumptions because of the rules that were followed to build the resulting artifacts. Hence, the method must be strict and rigid if properties such as completeness, correctness, realizability, and consistency are to be assured in the construction process. SBS itself is a rigid procedure to create a formal software requirement specification (SRS) [Car09, PP03]. Thus, the resulting artifact is consistent, complete, and traceably correct due the rules used for its creation. However, due to the fact that human error may occur during the application of SBS, an additional review afterwards helps to reduce this potential risk.

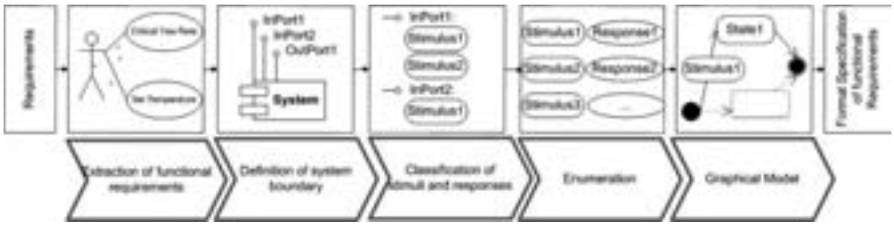


Figure 1: The manual steps in the Sequence-Based Specification process: Activities and artifacts.

The first step is the definition of the system boundary to derive all interfaces for input and output signals. Based on the requirements, all stimuli from these interfaces are determined. This comprises stimuli S that represent operations and responses R that are received as reactions from the system under development (SUD). A special stimulus is the empty event λ, which is also the starting point of the enumeration. In the automotive

domain, stimuli can originate in other electronic control units (ECU), in sensors on the same ECU, or in the system's environment.

In Figure 1, all required steps of the SBS process are shown:

- The functional requirements are identified and extracted. They are tagged for their traceability through the SBS and the following entire development.

- The SUD is considered as a Black Box. The SUD's system boundary is defined by its interfaces for information exchange.

- Stimuli (i.e. inputs to the SUD) and responses (i.e. information received from the SUD) are collected and classified. The classification is done on basis of the requirements w.r.t. the expected equivalent responses. The result is a set of classified inputs and outputs of the SUD.

- All possible stimulus sequences are completely and systematically enumerated and elongated. The method starts with the empty sequence λ of length zero, continuing with length one, two, and so on. Thus, it is assured that all possible combinations and permutations from the input domain are considered and related with the expected SUD's behavior. This enumeration process follows strict rules:

    - The stimulus extension is checked whether it is legal. Under certain conditions (e.g. when it is physically impossible to occur), an extension is considered as *illegal*. Then this sequence does not need to be considered anymore in the following iteration.

    - At each stimulus extension, the expected responses from the SUD are identified from the requirements and assigned to this stimulus sequence for later traceability.

    - After each stimulus extension, it is checked whether the expected behavior of the SUD w.r.t. all possible further extensions is equivalent to an already existing sequence. If such a sequence can be found, the current sequence can be mapped to the other one and the enumeration is continued for both in a common manner. This is called *reduction*.

- The process stops if all sequences are enumerated and elongated systematically, the corresponding responses are assigned for all sequences, and no further extension is possible.

It is to mention that each step – from the definition of the system boundary to the end of the enumeration – requires a reference to a requirement. Thus, the traceable correctness of the final SRS is assured. Furthermore, ambiguous or missing requirements are identified and all of them are validated w.r.t. their functional realizability.

All sequences that could not be reduced to another sequence are called *canonical sequences*. These sequences define the resulting state space for the SUD, and hence for the software implementation to realize the final system.

On the top layer, the states are considered as components of the SUD, as the states are on a high abstraction layer. Hence, the result is a first functional partitioning, from which the architecture and the deployment of components is derived. In the next step, the SBS is applied to each component, providing a design und implementation for it. The process of refinement is carried out recursively until a layer is reached on which the implementation is supplemented by manual coding or by model-based code generation e.g. from MATLAB/Simulink.

The first two steps are presented in Figure 2. SBS is used to derive an initial architecture of the SUD. On each component SBS is applied once again to generate a first design for each component. With the application of the SBS in each development step, properties such as completeness and traceable correctness can be assured down to an implementation artifact. Therefore, it is reasonable to adjust the development process to rely on the artifact SBS, which is described in the following section.

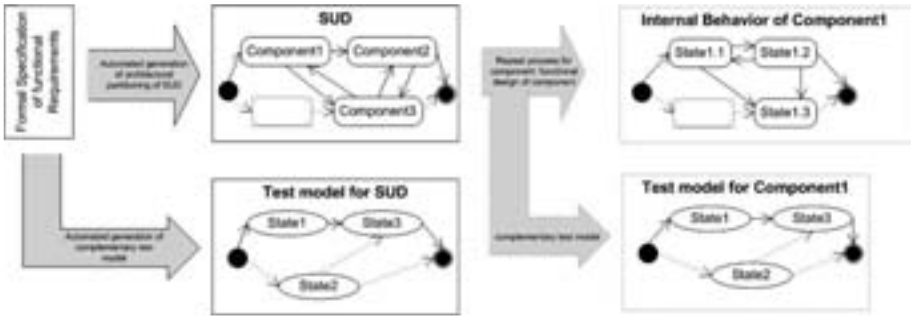


Figure 2: Recursive process for the usage of the SBS for the architecture and design.

From the artifact's point of view, the SBS process creates an executable state machine for the implementation as well as its complementary environmental model. The environmental model serves as a test model, from which any possible usage sequence can be derived, which serves as a valid and meaningful test case for using within MiL- or HiL-testing. In [Sie11], the development process for the test model with a systematic integration of time and timing requirements is described in greater detail. The executable state machine for the implementation can be simulated for validation and to assess the system's realizability. The executable state machine is equivalent to a Mealy automaton with further annotations.

The method of SBS can be applied to virtually any system that can finally be broken down to discrete signals and states. It demands skills to choose the appropriate level of abstraction, a suitable hierarchization, and to decide, what a state embraces. A decision has to be taken also on the question where SBS is applied to refine states and where

other methods, e.g. temporal logic, are used. This decision clearly depends also on the type of sub-functionality under consideration, which can be a reactive controller or logic.

4 Proposal for a Revised Development Process for Software-Intense Vehicle Functions

According to [Sel07] and which is also confirmed by countless projects, the later a failure is discovered the higher are the costs for identifying and fixing its fault. Moreover and also shown in the aforementioned publication, 40% of the top ten project risks ("4. requirements mismatch", "5. user interface[1] mismatch", "6. architecture", "7. requirements changes") are related to requirements, architecture, and design, and thus within the early stages during the development.

In Figure 3, the V-model is depicted which is annotated not only by the exponentially increasing costs to fix discovered faults but also by the mentioned project risks. Thus, it is obvious that the uniformly distributed effort over the project runtime that is suggested by the design of the V-model differs from the reality's facts; the risks are left-skewed but their fixing efforts and thus the related costs are right-skewed.

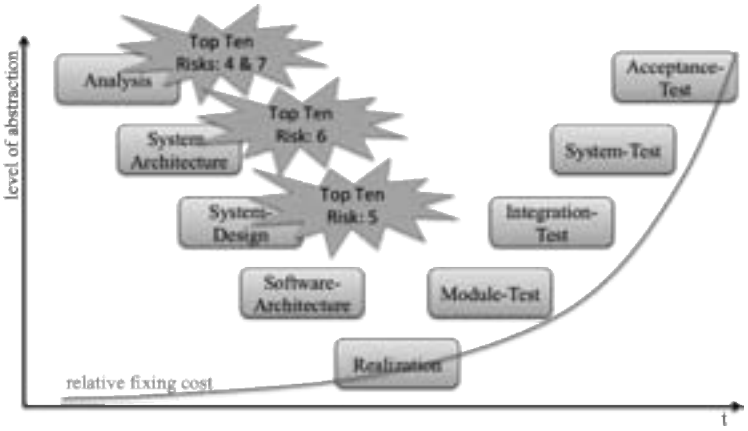


Figure 3: V-model, which is annotated by relative fixing costs and the 40% dominating project risks according to [Sel07].

However, any development approach, which is following this V-model, consists of consecutive steps, which depend on the previous one. This means in practice after finishing the requirements specification, the design and modeling of the architecture and the final implementation are started, which reveal sometimes inconsistencies or even realizability deficiencies in the specification. The subsequent step after the realization is the testing phase, which itself uses the specification again to unveil implementation errors. This step-wise and consecutive approach obviously moves the quality gate to the

[1] „user interface" includes not only system usage which is realized by an HMI but also technical interfaces which are used by neighbouring systems.

latest phase of the development where fixing inconsistencies and errors in a specification and implementation are most expensive. Furthermore, only implementation errors w.r.t. the specification are discovered but the quality of the specification itself like correctness, consistency, and realizability is hardly assessed.

But how can this widely realized [Cha05] but still accepted problem be overcome? Intuitionally, the quality gates must be moved to the earliest and reasonable phase during the development: For any system development project this is the point where the requirements specification is created and used to start the implementation. Obviously, any inconsistencies in this document will result in an erroneous implementation, which is only or even not in the worst case unveiled by test cases. Thus, the right-skewed distributed effort and costs must be corrected to result in a left-skewed distributed effort with a strong focus on the actual requirements specification in the best case.

Therefore, a combined method to act as quality gate, which serves both to validate the requirements specification and to support the implementation, is necessary. This method shall therefore provide consistent and complete test cases to reduce the test effort on the one hand; on the other hand, the method could also provide implementation artifacts directly. Thus, this method serves as so-called single-point-of-truth (SPoT) which must formally guide the process of scrutinizing and validating the requirements and their interrelations but which also must act as source model for generating implementation artifacts or test cases according to a project's specific needs.

At this point, the method of constructive requirements modeling as outlined in the aforementioned section comes into play. The constructed formal model, which fundamentally is an annotated Mealy automaton, contains by principle all valid usage scenarios for a system. As outlined, this is achieved by a systematic enumeration of the system's input signals combined with the expected system reaction according to the initial requirements specification. Thus, the requirements specification is validated iteratively in terms of consistency, correctness, and realizability during the analysis stage in a guided manner.

Due to the fact that each state in the resulting Mealy automaton also contains the expected system's output, the formal requirements model can serve either as source model to generate first implementation artifacts on the one hand, or to provide input signal sequences to derive only valid and meaningful test cases on the other hand. Their usage is shown in Figure 4, which results in a revised and slightly left-skewed variant of the V-model. These modifications are strongly encouraged by the results from [SHG10] and [SHGB11].

Even if the V-model is widely used in the automotive domain for the development of vehicle functions, the outlined method can also be used with modern agile methods in iterative contexts. This is especially interesting when providing test cases for MiL testing as described in [SHGB11]. Thus, prototypical implementation models can iteratively be improved.

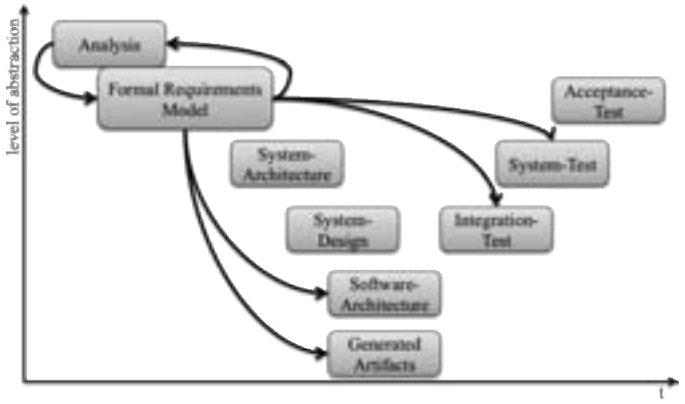


Figure 4: Revised V-model with a strong focus on a formal requirements specification model from which implementation and testing artifacts are generated automatically to reduce implementation and testing efforts.

In practice, each transition of the Mealy automaton can also be weighted with a probability to reflect more likely usage profiles and to direct the test case generation [SL10]. Furthermore, these generated test cases can be directly used during a manual prototyping phase to enable and support a "test first" approach for MiL-testing, while getting further insights from the prototype. In the end, these requirements models enable a quality gate at the very earliest time point to save valuable resources later on in the V-model development process. Furthermore, artifacts from suppliers can be validated objectively and even automatically.

5 Case Study: Auxiliary Heating System

In this section, the previously described revised development process was applied during the software development of an auxiliary heating system. These systems are mainly used in road vehicles to heat the passenger compartment and to defrost the windows. In hybrid or electric vehicles, their usage is extended for components of power transmission and power supplies.

In the following, the conventional system for the passenger compartment is considered. Depending on the customer's choice of a temperature, the auxiliary heating system determines whether it is necessary to heat or to cool. Based on the environmental conditions it starts at a calculated time with its activity. During the phase of heating or cooling it is monitoring the environment as well as the compartment w.r.t. changes of the temperature. If necessary, it adapts its activity to the changed conditions. The system contains both discrete and continuous behavior.

Furthermore, in recent vehicles there are a lot of possibilities to control the auxiliary heating system and air conditioning. It is possible to program timers via the key, the

Multimedia Interface (MMI), or directly at the climate control. Moreover, specific settings can be overridden by user actions that have higher priority.

5.1 Application of the SBS to the Auxiliary Heating System

As outlined in the revised V-model development process, we applied the SBS already at the very early stage to derive a formal requirement specification from initial requirements of the auxiliary heating system. A high functional level of abstraction was chosen as the purpose of the model was also the derivation of test cases for integration testing. Therefore, different sources had to be brought into the process to formalize the requirements, e.g. contents from a requirement management system, sketches and minutes from meetings, and also experts for the functionality who were in charge of the final product. After the analysis and clarification of the requirements, the models as described in Section 3 could be created.

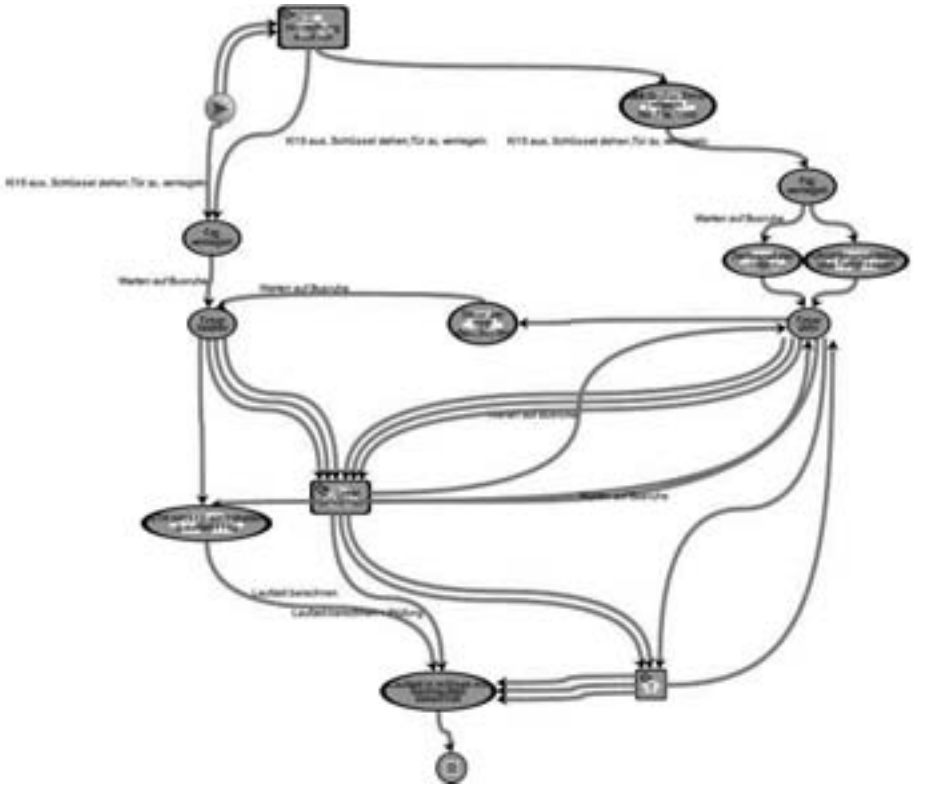


Figure 5: Detailed model for the auxiliary heating functionality. This model was used during integration testing on the ECU.

To give an idea of the size and complexity, the following figures of the model for the auxiliary heating system are given: On the top layer the model consists of five states. Three of them, i.e. all except the start and final state, are refined. As a result, the

implementation consists of three components that interact with each other. On the next layer the SBS process is applied to each component. The result consists of 39 states with 52 transitions. These figures provide quite a good idea of the range, in which the size of the model diagrams resulted.

As described in Section 3, the internal behavior and the appropriate test model are derived from the SRS model. In Figure 5, a detail of the test model is depicted, which was derived on a higher abstraction layer for integration testing on the final ECU. The abstraction on this layer is determined by the interfaces provided through the target ECU and its bus interfaces. For the sake of clarity, the states are grouped. The rectangular boxes in Figure 5 are grouped states, in which possible behavior with the same functional purpose is specified. Furthermore, the grey and white states are further refined. These states can be entered and the internal behavior of the state is specified in a new sub-diagram.

The process was applied until a level of abstraction was reached which was sufficiently detailed. The complete automaton, i.e. the top model including all states and transitions from the refinements, consists of 796 states and 1,019 transitions. The average length of one usage sequence is 66 transitions, which assumes an operational profile with uniformly distributed probabilities.

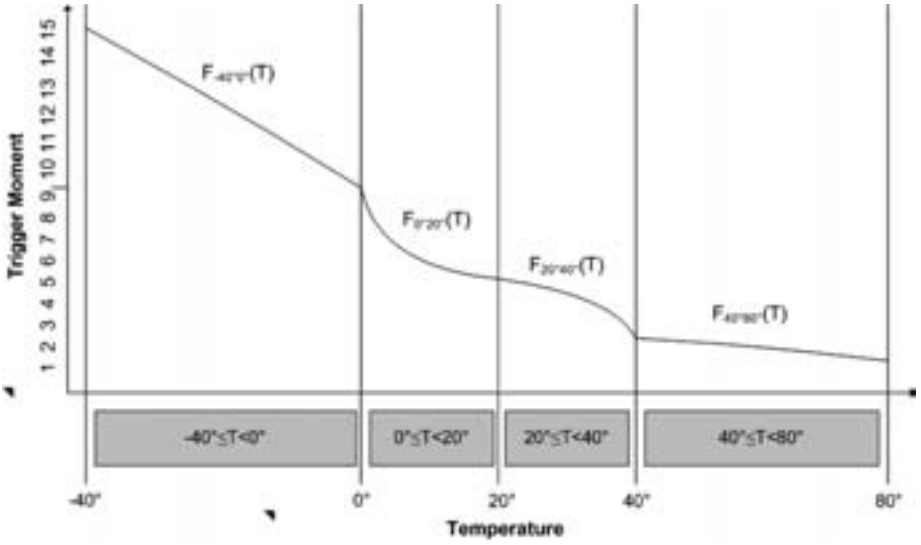


Figure 6: Example of continuous functions that are mapped to discrete states during the SBS process.

The system's continuous behavior is mapped to discrete states. Continuous functions are used to specify the values between discrete states. In Figure 6, the function e.g. F0°20°(T) provides all continuous values between temperatures including 0° and excluding 20°. The abscissa represents the temperature in degree Celsius and the ordinate describes the trigger momentum for an actuator in the heating system. As illustrated in that figure, the continuous transition between discrete states is specified

with functions, and the discrete states serve to span the state space of the automaton and for e.g. the calculation of metrics.

5.2 Results

As expected and described in Section 4, ambiguities in the requirement specification could be identified and solved during the formalization process. The deficiencies were apparently related to the negligence of usage scenarios, which were represented by systematically enumerating possible stimuli sequences that might occur in the field use. Thus, error-prone interpretations during the implementation could be avoided in a systematic manner and test engineers already had a high confidence in the realization. Furthermore, they could easily derive test cases for their model-based testing process directly from the SRS model.

As there are many phases of the development process between the construction of the models and the testing of the integrated system, a variety of potential errors can be detected. In between are tool changes, code processing, and additional basic software modules in the final ECU for e.g. the I/O communication to the SUT.

In contrast to the traditional V-model-based development process, informal specification documents did not provide clear, complete, and consistent answers to all expected usage scenarios for the auxiliary heating system. Thus, the responsible engineers for the functionality were involved to clarify these issues. Thus, the initial requirements could be revised thoroughly and systematically to get a complete and consistent formal model.

As already mentioned, these resulting SRS models also served to improve the testing of the implementation. Additional errors could be identified, that had not been discovered before. They were mainly related to neglected possible usage scenarios, which had not been tested and on the other hand, to deficiencies in the requirement specification, that led to vagueness in the interpretation on the implementation and test side. The acceptance of the users was high because finally the created model served as basis for discussion and communication. Encouraged by the results of this project, the revised V-model based development process shall be applied for new functionalities on a project for thermo-management.

6 Conclusion and Outlook

Formal methods are not the one and only Holy Grail to tackle the increasing complexity of upcoming software-intense systems; using a formally derived artifact at the very early stage does not automatically mean that it specifies the “right” requirements. However, and that is the important aspect, it is assured that all following stages rely on a consistent, complete, and realizable model, which serves as the single-point-of-truth in an artifact-driven development process, and thus, error-prone human interpretations can be significantly reduced or even avoided.

Nevertheless, formal methods contribute mainly to reduce the probability to discover unexpected errors during later stages in traditional development processes. As shown in [SHGB11], even manageable embedded systems with a sophisticated testing process at first sight did benefit from the SBS approach – already at early stages by supporting the MiL-testing. As a consequent step, this contribution outlines a revised V-model based development process, which explicitly embraces a method to constructively model requirements in a formally consistent, complete, and realizable manner at the earliest possible stage. Its applicability and benefits were demonstrated on the example of the requirements elicitation and development of an auxiliary heating system.

In the future, formal methods need to be rather mandatory and not only limitedly recommended as stated in the ISO-26262 standard due to two reasons: At first, the abandonment would provoke the risk of lately discovered errors which are very expensive in terms of product recalls for example; second, the customer expects a reliably and thoroughly engineered and operating software-driven system. However, future work must be carried out to investigate the expressiveness of applicable formal methods for requirements specification regarding the type of embedded systems in terms of continuous signal processing or event-driven system. Moreover, the underlying characteristics of a system's input signals should be investigated to provide guidelines for applicability and to derive best practices. Furthermore and as already mentioned as concerns in Section 1, appropriate and user-friendly tooling is necessary to increase the overall acceptance among developers and for test and requirements engineers.

References

[BBHR07] C. Basarke, C. Berger, K. Homeier, and B. Rumpe, "Design and Quality Assurance of Intelligent Vehicle Functions in the 'Virtual Vehicle'," *Proceedings of 11. Automobiltechnische Konferenz – Virtual Vehicle Creation, Stuttgart*, vol. 9, 2007.

[BBKR09] A. Bartels, C. Berger, H. Krahn, and B. Rumpe, "Qualitätsgesicherte Fahrentscheidungsunterstützung für automatisches Fahren auf Schnellstraßen und Autobahnen." In: *Proceedings des 10. Braunschweiger Symposiums "Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel,"* 2009, vol. 10, pp. 341-353.

[Ber10] C. Berger, "Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles." Aachen, Germany: Shaker Verlag, *Aachener Informatik-Berichte*, Software Engineering Band 6, 2010.

[BR12] C. Berger and B. Rumpe, "Engineering Autonomous Driving Software." In: *Experience from the DARPA Urban Challenge*, C. Rouff and M. Hinchey, Eds. London, UK: Springer, 2012, pp. 243-271.

[Car09] J. M. Carter, "Sequence-Based Specification of Embedded Systems." University of Tennessee, Knoxville, 2009.

[Cha05] R. N. Charette, *"Why Software Fails."* In: IEEE Spectrum, Vol. 42, no. 9 (INT), 2005, p. 36.

[CJL+08] D. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg, F. Törner, and M. Törngren, "Modelling Support for Design of Safety-Critical Automotive Embedded Systems," *SafeComp*, vol. 5219, Series: Computer Safety, Reliability, and Security (SAFECOMP), pp. 72-85, 2008.

[EN01] EN 50128 – Railway applications – Software for railway control and protection, November 2001.

[DO11] DO 178C – Software Considerations in Airborne Systems and Equipment Certification, January 2012.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston, “Experience with formal methods in critical systems.” In: *IEEE Software*, vol. 11, no. 1, pp. 21-28, 1994.

[GNA+12] J. M. Carrillo de Gea, J. Nicolás, J. L. Fernández Alemán, A. Toval, C. Ebert, A. Vizcaíno, “Requirements Engineering Tools: Capabilities, Survey and Assessment.” In: *Information and Software Technology*, 2012.

[ISO11] ISO-26262 – Road Vehicles – Functional Safety, November 2011.

[KBP01] C. Kaner, J. Bach, and B. Pettichord, “Lessons Learned in Software Testing.” Wiley-India, 2001.

[Kon11] B. B. Konka, “A case study on Software Testing Methods and Tools,” University of Gothenburg & Chalmers University of Technology, July 2011.

[PP03] S. J. Prowell and J. H. Poore, “Foundations of sequence-based software specification.” In: *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 417-429, 2003.

[PBKS07] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, “Software Engineering for Automotive Systems: A Roadmap.” In: *Proceedings of the 2007 Future of Software Engineering*, 2007, pp. 55-71.

[PHP11] A. Post, J. Hoenicke, and A. Podelski, “Vacuous real-time requirements,” in 2011 IEEE 19th International Requirements Engineering Conference, 2011, pp. 153–162.

[RBK06] B. Rumpe, C. Berger, and H. Krahn, “Software Engineering Methods for Quality Management of Intelligent Automotive Systems.” In: *Integrierte Sicherheit und Fahrerassistenzsysteme*, 2006, no. 1960, pp. 473-486.

[Sel07] R. Selby, “Software Engineering: Barry W. Boehm’s Lifetime Contributions to Software Development, Management, and Research (Practitioners).” Wiley-Blackwell, 2007.

[Sie11] S. Siegl, “Specification and verification of the embedded real-time systems in the automotive domain.” Göttingen, Germany: Cuvillier Verlag, *Audi Dissertationsreihe*, Band 52, 2011.

[SHG10] S. Siegl, K.-S. Hielscher, and R. German, “Model Based Requirements Analysis and Testing of Automotive Systems with Timed Usage Models.” In: *Proceedings of the 18th IEEE International Requirements Engineering Conference (RE’10)*, 2010, pp. 345-350.

[SHGB11] S. Siegl, K.-S. Hielscher, R. German, and C. Berger, “Formal Specification and Systematic Model-Driven Testing of Embedded Automotive Systems.” In: *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE 2011)*, 2011, pp. 1530-1591.

[SL10] S. Siegl and C. Lauer, ”Design and Validation of Embedded Real-Time Applications.” In: *Proceedings Of Embedded Real Time Software and Systems (ERTSS)*, Toulouse, France, 2010.

Increasing the reliability of single and multi core systems with software rejuvenation and coded processing

Juergen Braun[1], Juergen Mottok[1], Christian Miedl[2], Dirk Geyer[2], Mark Minas[3]

[1] - University of Applied Sciences Regensburg, LaS³,
Seybothstraße 2, D-93053 Regensburg,
{Juergen.Braun;Juergen.Mottok}@hs-regensburg.de
[2] - AVL Software & Functions GmbH,
Im Gewerbepark B27, D-93059 Regensburg,
{Christian.Miedl;Dirk.Geyer}@avl.com
[3] - Universität der Bundeswehr München,
D-85577 Neubiberg,
Mark.Minas@unibw.de

Abstract: The safety of electric vehicles has the highest priority because it helps contribute to customer confidence and thereby ensures further growth of the electromobility market. Therefore in series production redundant hardware concepts like dual core microcontrollers running in lock-step-mode are used to reach ASIL D safety requirements given from the ISO 26262.

Coded processing is capable of reducing redundancy in hardware by adding diverse redundancy in software, e.g. by specific coding of data and instructions. A system with two coded processing channels is considered. One channel is active and one is in cold standby. When the active channel fails, the service is switched from the active channel to the standby channel. It is imaginable that the two channels with implemented coded processing are running with time redundancy on a single core or on a multi core system where for example different ASIL levels are partitioned on different cores.

In this paper a redundant concept based on coded processing and software rejuvenation will be taken into account.

1. Introduction

Nowadays the importance of the reliability in safety-critical and life-critical systems is well recognized [Bao05]. Redundancy does not only mean the duplication of systems. By definition, all additional units and methods are included, that are implemented for error detection and error avoidance. Such methods are for example integrated test units as well as error-detecting codes and coded processing, like Safely Embedded Software (SES) [Mot12]. With embedded multicore microcontrollers software aging [Par94] is getting a more and more important point even for embedded systems. Thereby the hardware fault tolerance is well understood since many years, but the software fault tolerance is often the origin of most of the reliability problems. In a standard or low performance embedded system there are nearly no dynamic factors, like the static memory allocation, but in high performance or future embedded systems a dynamic

behaviour will be necessary. At the moment these issues have less relevance for safety-critical systems. For instance, in the case of software that undergoes a safety certification process, dynamic memory management is typically avoided [Cor11]. A preventive and proactive solution to software aging is software rejuvenation involving the restoration of a system to a clean internal state [Sar09]. The proactive fault management has to be added as a complementary approach in addition to the traditional reactive recovery mechanisms with subject to aging. At a certain time with the proactive fault management a rejuvenation operation is scheduled, so that potential error conditions could be removed. There are two different approaches for software aging and rejuvenation. The measurement based are statistical analysis for predicting a time window where rejuvenation should be prefered [Cas01] [Gar98] [Li02] [She03] [Vai99]. An optimal rejuvenation time could not be determined with this statistical approach. The analytical approach is done with a Markov process or with stochastic Petri net models. With this models it is possible to compute the system availability [Doh00d] [Doh00n] [Gar95] [Hua95] [Vai01]. During my research studies for coded processing I have been inspired by the work [Kou10] for the following considerations to proof with a Continuous Time Markov Chain that software rejuvenation between different coded processing channels improves the availability of the embedded system significantly, compared with an embedded system without any rejuvenation actions and without coded processing.

The paper is structured as follows. After the presentation of the principle of Coded Processing based on the example of Safely Embedded Software in section 2, an introduction of Software Rejuvenation is given in section 3. In section 4 the redundant systems without and with Safely Embedded Software in combination with Software Rejuvenation are modelled, which are the basis for section 5 in which the Mean Time To Failure is calculated with the help of a Continuous Time Markov Chain. Section 6 contains some conclusions and an outlook for future work.

2. Coded Processing

The concept of coded processing is capable of reducing redundancy in hardware by adding diverse redundancy in software, e.g. by specific coding of data and instructions. Hardware and software coding can be combined using approaches like the Vital Coded Processor [For89]. Consequently besides the actual safety-critical control program, also the other programs can run on the same hardware. Thus it is possible to specifically protect only the safety-critical parts of the control program. Coded Processing enables the verification of safety properties and fulfills the condition of single fault detection [Mot12]. Coded Processing does not constrict capabilities but rather supplements multi-version software fault tolerance techniques like N version programming, consensus recovery block techniques, or N self-checking programming [Mot12]. In this paper the Safely Embedded Software approach is described in more detail.

2.1. Safely Embedded Software (SES)

The given SES approach generates the safety of the overall system in the application software level. SES is based on the (AN+B)-code of the Coded Monoprocessor transformation of original integer data x_f into diverse coded data x_c [Mot12].

Coded data fulfills this relation:

$$x_c = A \cdot x_f + B_x + D_t \text{ where } \begin{matrix} x_c, x_f \in \mathrm{Z}, A \in \mathrm{N}^+, B_x, \\ D_t \in \mathrm{N}_0, B_x + D_t < A \forall x, t \end{matrix} \quad (1)$$

The prime number A is important for safety characteristics like Hamming Distance and residual error probability $P_{re} = 1/A$ of the code [Mot12]. Hamming distance gives information about the maximum number of fully detectable bit errors and residual error probability P_{re} is the probability that the corruption of the data remains undetected after performing the decoding procedure [Sch06]. Number A has to be prime because in case of a sequence of i faulty operations with constant offset f, the final offset will be $i \cdot f$. If A is not a prime number then several factors of i and f may cause multiples of A. If a prime number is used as A, this offset is only a multiple if i or f is divisible by A. This is the same fact for the multiplication of one or two faulty operands. Additionally, so called deterministic criteria like the above mentioned Hamming Distance and the Arithmetic Distance must be considered for choosing an adequate prime number. Other functional characteristics like necessary bitfield size and consequential handling of overflow are also caused by the value of A. The simple transformation $x_c = A \cdot x_f$ is illustrated in Fig. 1.

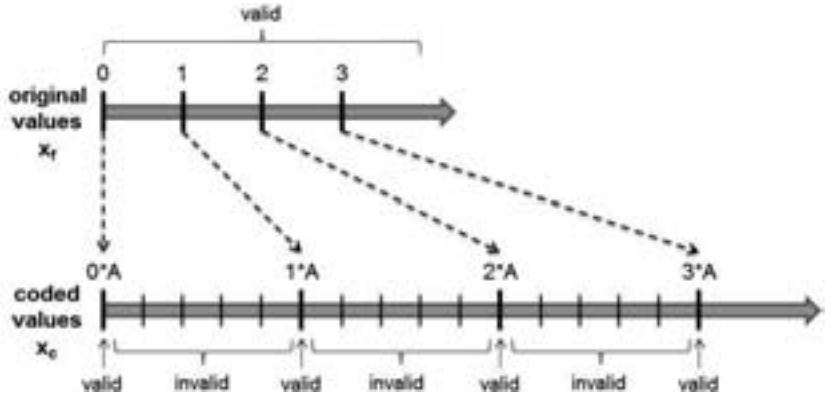


Figure 1: **Simple transformation for $x_c = A * x_f$.**

To ensure the correct memory addresses of the variables any user defined specific number or the memory address of the variable itself could be used as static signature B_x [Mot12]. The dynamic signature D_t ensures that the variable is used for example in the correct task cycle or function call.

The two software channels (original data and coded data) could be verified for example at the end of each task cycle before starting the output of the calculated values. Therefore the instructions are coded in such a way that, either a comparator could verify the diverse channel results for the condition $z_c = A \cdot z_f + B_z + D_t$, or the coded channel could be checked directly by the verification condition $(z_c - B_z - D_t) \bmod A = 0$ (cf. (1)).

3. Software Rejuvenation

Software rejuvenation is a technique which has been first proposed by Huang et. al. in 1995 [Hua95]. Since this time many papers have been published [Cor11] to characterize and mitigate the Software Aging phenomenon. In long-running operational software, due to software aging, the accumulation of errors leads for example to progressive resource depletion and finally to the crash of the system [Bao05]. Software rejuvenation is a fault reaction technique to prevent that failures in continuously running systems occur. At certain points in the program flow the currently used variables and the corresponding program counter will be stored on a stack. If an error is detected the rejuvenation could switch the service from the active coded channel to the standby coded channel. Consequently the execution of the function could be continued at this point in the program flow using the other channel. This involves stopping the task of one coded channel and switching the task to the other coded channel which was in standby. Due to this environment diversity the transient failures in software could be corrected, because the coded channel which is in standby could be restarted in an internal state with a cleaned environment like the memory. The failure triggered rejuvenation is reactive, so that the action is started after a failure. In addition the software rejuvenation process can also be started automatically due to measurements or calculations which determine the best time window for switching. This means that the action is startet preventive and proactive. The likelihood of successfully completing the current task will be increased by the following steps. The task will be switched periodically from one coded channel to the second coded channel, while restarting the task at a previous checkpoint and in the meantime the rejuvenation process will be started on the currently inactive channel. It must be stressed that the periodical switching does not replace the switching due to an error. This mechanism complements the switching due to an error, so that this case becomes less likely in the system. When errors occur in the system, it is still necessary to trigger the software rejuvenation spontaneously.

Software rejuvenation does not solve the root cause of the failures, consequently the software aging will continue so that the software rejuvenation has to be executed cyclically. Since software aging leads to transient failures in systems, environment diversity can be employed pro-actively to prevent degradation or crashes [Vai99]. Such errors are transient and they are non-deterministic or difficult to characterize. Consequently these errors could occur after a long time without disturbing the proper execution of the application before. Even now, software aging has been observed in specialized software used in high-availability and safety-critical applications [Hua95].

The main aim of SES in the case of rejuvenation is to detect errors reliable. After the detection of an error the switching to the standby unit is triggered and the rejuvenation is started. Beside this the software rejuvenation is triggered periodically, so that after every rejuvenation interval the task is restarted at a clean internal state. Due to this the reliability of the system could be increased.

4. Modelling of the redundant system

Two models are discussed in this paper. The first model is a redundant system without software rejuvenation and the second model is with coded processing and software rejuvenation.

In this paper with software rejuvenation, always the first-level rejuvenation or partial rejuvenation is meant. In the literature there are first-level and two-level rejuvenation models available. First-level rejuvenation stopps only certain tasks of the system and switches them to a redundant system. However the two-level rejuvenation, also called full rejuvenation, could stop all running tasks and restarts the system. With two-level rejuvenation depending on the condition of the software environment the different rejuvenation actions (full or partial) will be triggered [Jin05][Kou05][Xie05]. This full rejuvenation is in the automotive environment for the most systems not possible, because the demanded reaction times are shorter than the time needed for the restart of the whole system. For systems in the automotive environment a certain availability must be guaranteed for the whole driving cycle.

Consequently in this paper with software rejuvenation always the first-level rejuvenation is meant.

4.1. Markov Chain

The Markov chain is a stochastic process and named after Andrey Andreyevich Markov [Hil07]. It consists of a finite number of states and represents the possible transitions from one state to another. In the case of a Markov chain of first order, for each step it does not matter what has been the previous sequence of steps. For determining the next step it is only relevant which is the current state. Due to this fact the process is called "memoryless". This is called the Markov property. The aim is to be able to calculate the probability for the occurrence of certain states in the future.

With Markov chains it is possible to describe any redundant system. The states of the Markov chain correspond to the possible system states, which represent the functional or failure states of the components of the system. At a certain time the system is always in a certain state of the Markov chain. Consequently all failure rates as well as their appearance can be taken into account.

Each state is represented by a circle with a number in it. To improve the readability the states could be additionally named, like “no channel failed”. The transitions between the states are represented by arrows from the destination state to the target state. The arrows

are labeled with the transition rate, which represents the rate that the system goes from this state to a certain state.

Electronic systems behave as if they would fail nearly at any time with the same failure rate λ. There is an exception for the early failures and for failures due to aging [Bra12]. Based on these facts the traditional Weibull distribution is unable to model the complete lifetime of electronic systems. Therefore the extended Weibull distribution has been developed which matches such systems with a bathtub-shaped failure rate function. The characteristics of the extended Weibull distribution are, that at the beginning the Weibull distribution is dominating the gradient of the curve, at the end the function is increasing rapidly according to a nominal distribution and in between the failure rate is almost constant which corresponds to exponentially distributed random failures.

For simplification an ideal model is assumed for electronic systems [Bör04]. In this ideal model the early and late failures are neglected.

4.2. System with Cold Redundancy without SES

A system with two channels is considered. One channel is active and one is in cold standby. Two channels are running on a multi core system where for example different ASIL levels are partitioned on different cores. The important point is that one channel is active and one is in standby. When the active channel fails, the service is switched from the active channel to the standby channel.

As shown in Fig. 2 we assume, that the active channel fails with constant failure rate λ_1. At the beginning no channel failed, so that the system is in state 1, where one channel is active and the other channel is in cold standby. At this point no automatic switching at a certain time is implemented. Consequently only in the case of an error the system switches from the active channel to the standby channel and enters state 2. The system enters state 3 with failure rate λ_1, if another error occures. In state 3 the system must enter the error state.

There is also the possibility that a failure occurs in both channels. In this case with rate λ_2 starting from state 1 the system must enter state 3 directly without traversing state 2.

Consequently there is no differentiation in the error state. This means that the error state will be entered for example after a common cause failure, just as well after the occurrence of transient errors due to electro migration.

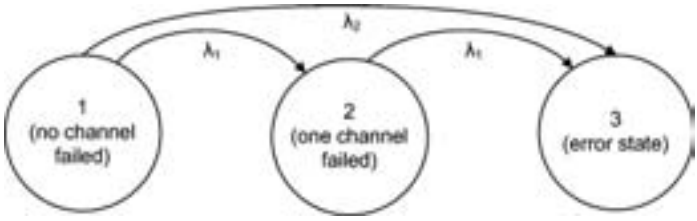


Figure 2: **system with cold redundancy (no rejuvenation)**

4.3. Software Rejuvenation on a System with Cold Redundancy with SES

Again a system with two channels is considered. In this model each channel is coded with SES wherupon one channel is active and one is in cold standby. The system switches from the active channel to the standby channel in the case of an error as well as cyclically at the time when the cyclic rejuvenation is started. Consequently there is an automatic switching implemented at a certain time. As shown in Fig. 3, at the beginning no channel failed, so that the system is in state 1, where one channel is active and the other channel is in cold standby.

In the case of an error the system switches from the active channel to the standby channel and enters state 2. The system will enter state 4 where the rejuvenation will be started or the system will enter state 3, if another error occures during the checks of the failed channel. In state 3 the system must enter the error state.

Additionally the rejuvenation process will be triggered cyclic. For the cyclic rejuvenation process the system switches from the active channel to the standby channel, so that the rejuvenation process could be started for the channel, which was the active channel before and enters state 5. After the rejuvenation process, the coded channel is available in cold standby again.

Compared to the previous model without sofware rejuvenation, also the case where software rejuvenation can fail has to be considered. If the rejuvenation process (triggered due to an error or cyclically) is not completed properly or is performed improperly, the state 3 will be entered to bring the system into the error state. This is the result, because the failed rejuvenation is a failure state, which indicates an abnormal function. The consequence of this is, that the system fails in the same manner as before, if a failure occurs on the active channel during the software rejuvenation process. Because of the ongoing software rejuvenation process, no switching to the standby channel is possible.

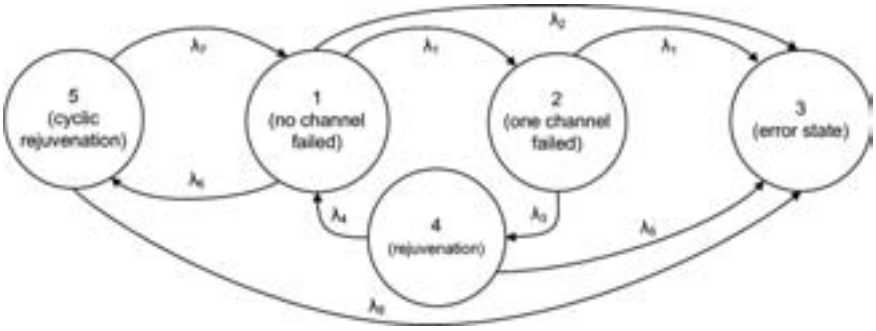


Figure 3: system with coded channel in cold redundancy and rejuvenation

5. Continuous Time Markov Chain (CTMC)

With the help of Markov-models it is possible to calculate reliability values and safety parameters like point-availability, availability, MTTF and reliability. The rates for the state transitions can be specified for each transition from one state to another as well as the probability that the system remains in the state. Constant failure rates are assumed and an exponential distribution. Thus the transition rates are always $\lambda \geq 0$.

5.1. Introduction

Normally the reliability problems are concerned with systems which consist of a number of discrete and identifiable states and continuous in time. A stochastic chain with a continuous parameter space is called markov chain iff. for all $n \geq 0$, each sequence $t_0 < t_1 < ... < t_n < t$:

$$P(X(t) = x \mid X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, ..., X(t_0) = x_0) \\ = P(X(t) = x \mid X(t_n) = x_n) \tag{2}$$

The transition matrix Q represents the transition rates for all n states:

$$Q = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ q_{n1} & q_{n2} & \cdots & q_{nn} \end{bmatrix} = \begin{bmatrix} 1-A_1 & \lambda_{12} & \cdots & \lambda_{1n} \\ \lambda_{21} & 1-A_2 & \cdots & \lambda_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \lambda_{n1} & \lambda_{n2} & \cdots & 1-A_n \end{bmatrix} \tag{3}$$

The row sum has to be 1. If there is no transition from one state to another the transition rate is 0. The elements of the matrix described with q_{ij} represent the transition rate from state *i* to state *j* where $i, j \in [1...n]$. Therefore each row represents all transitions from state i to another state and each column represents all transitions from another state to state *j*.

For example the matrix *Q* could be subtracted from the identity matrix *I* to get the matrix *M*:

$$M = I - Q = \begin{bmatrix} A_1 & -\lambda_{12} & \cdots & -\lambda_{1n} \\ -\lambda_{21} & A_2 & \ddots & -\lambda_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ -\lambda_{n1} & -\lambda_{n2} & \cdots & A_n \end{bmatrix} \tag{4}$$

Where the exit rate A_i of state *i* is defined as:

$$A_i = \sum_{k=1, k \neq i}^{n} \lambda_{ik} \tag{5}$$

5.2. System with Cold Redundancy without SES

Based on the Markov chain Fig. 2 defined in chapter 4.2 in combination with (3) and (5) the matrix *Q* could be defined for a system with cold redundancy:

$$Q = \begin{bmatrix} 1-(\lambda_1+\lambda_2) & \lambda_1 & \lambda_2 \\ 0 & 1-\lambda_1 & \lambda_1 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

The matrix *G* contains only of state 1 and 2, because the system is fully operational in these states.

$$G = \begin{bmatrix} 1-(\lambda_1+\lambda_2) & \lambda_1 \\ 0 & 1-\lambda_1 \end{bmatrix} \tag{7}$$

For the mean time to failure (MTTF) calculation, which measures the average time to failures, the matrix *M* is calculated:

$$M = I - G = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1-(\lambda_1+\lambda_2) & \lambda_1 \\ 0 & 1-\lambda_1 \end{bmatrix} = \begin{bmatrix} \lambda_1+\lambda_2 & -\lambda_1 \\ 0 & \lambda_1 \end{bmatrix} \tag{8}$$

With this matrix *M* the matrix *N* could be calculated:

$$N = [M]^{-1} = \begin{bmatrix} \lambda_1+\lambda_2 & -\lambda_1 \\ 0 & \lambda_1 \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{\lambda_1+\lambda_2} & \frac{1}{\lambda_1+\lambda_2} \\ 0 & \frac{1}{\lambda_1} \end{bmatrix} \tag{9}$$

For the calculation of MTTF value, the first row of the matrix *N* has to be summed up:

$$MTTF = \frac{1}{\lambda_1+\lambda_2} + \frac{1}{\lambda_1+\lambda_2} \tag{10}$$

The following values are assumed:

$$\lambda_1 = 10^{-7}\,\frac{1}{h} \tag{11}$$

$$\lambda_2 = \beta \cdot \lambda_1 = 0.05 \cdot 10^{-7}\,\frac{1}{h} \tag{12}$$

For the determination of the transition rate λ_1 the ISO 26262 [ISO11] was used. In the ISO 26262 the random hardware failure rate of ASIL B and ASIL C systems is defined as $10^{-7}\,h^{-1}$. According to the IEC 61508 part 6 [IEC00] a β-factor is used, to calculate the rate λ_2. This is the rate of a failure of the safety-related system due to a common cause failure. For the system without SES the given β-factor for redundant systems with poor diagnostic tests has been choosen from IEC 61508.

Consequently with equation 10 to 12 the MTTF could be calculated:

$$MTTF \approx 2{,}170\,y \tag{13}$$

5.3. Software Rejuvenation on a System with Cold Redundancy with SES

Again based on the Markov chain Fig. 3 defined in chapter 4.3 in combination with (3) and (5) the matrix Q could be defined for a system with cold redundancy and software rejuvenation:

$$Q = \begin{bmatrix} 1-(\lambda_1+\lambda_2+\lambda_6) & \lambda_1 & \lambda_2 & 0 & \lambda_6 \\ 0 & 1-(\lambda_1+\lambda_3) & \lambda_1 & \lambda_3 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \lambda_4 & 0 & \lambda_5 & 1-(\lambda_4+\lambda_5) & 0 \\ \lambda_7 & 0 & \lambda_8 & 0 & 1-(\lambda_7+\lambda_8) \end{bmatrix} \tag{14}$$

The matrix G contains only of state 1, 2, 4 and 5, because in these states the system is fully operational.

$$G = \begin{bmatrix} 1-(\lambda_1+\lambda_2+\lambda_6) & \lambda_1 & 0 & \lambda_6 \\ 0 & 1-(\lambda_1+\lambda_3) & \lambda_3 & 0 \\ \lambda_4 & 0 & 1-(\lambda_4+\lambda_5) & 0 \\ \lambda_7 & 0 & 0 & 1-(\lambda_7+\lambda_8) \end{bmatrix} \tag{15}$$

For the mean time to failure (MTTF) calculation, which measures the average time to failures, the matrix M is calculated:

$$M = I - Q = \tag{16}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 1-(\lambda_1+\lambda_2+\lambda_6) & \lambda_1 & 0 & \lambda_6 \\ 0 & 1-(\lambda_1+\lambda_3) & \lambda_3 & 0 \\ \lambda_4 & 0 & 1-(\lambda_4+\lambda_5) & 0 \\ \lambda_7 & 0 & 0 & 1-(\lambda_7+\lambda_8) \end{bmatrix} =$$

$$= \begin{bmatrix} (\lambda_1+\lambda_2+\lambda_6) & -\lambda_1 & 0 & -\lambda_6 \\ 0 & (\lambda_1+\lambda_3) & -\lambda_3 & 0 \\ -\lambda_4 & 0 & (\lambda_4+\lambda_5) & 0 \\ -\lambda_7 & 0 & 0 & (\lambda_7+\lambda_8) \end{bmatrix}$$

With this matrix M the matrix N could be calculated:

$$N = [M]^{-1} = \begin{bmatrix} (\lambda_1+\lambda_2+\lambda_6) & -\lambda_1 & 0 & -\lambda_6 \\ 0 & (\lambda_1+\lambda_3) & -\lambda_3 & 0 \\ -\lambda_4 & 0 & (\lambda_4+\lambda_5) & 0 \\ -\lambda_7 & 0 & 0 & (\lambda_7+\lambda_8) \end{bmatrix}^{-1} = \tag{17}$$

$$= \det(M)\cdot \begin{bmatrix} (\lambda_1+\lambda_3)(\lambda_4+\lambda_5)(\lambda_7+\lambda_8) & \lambda_1(\lambda_4+\lambda_5)(\lambda_7+\lambda_8) \\ \lambda_3\lambda_4(\lambda_7+\lambda_8) & (\lambda_1+\lambda_2+\lambda_6)(\lambda_4+\lambda_5)(\lambda_7+\lambda_8)-\lambda_6(\lambda_4+\lambda_5)\lambda_7 \\ (\lambda_1+\lambda_3)\lambda_4(\lambda_7+\lambda_8) & \lambda_1\lambda_4(\lambda_7+\lambda_8) \\ (\lambda_1+\lambda_3)(\lambda_4+\lambda_5)\lambda_7 & \lambda_1(\lambda_4+\lambda_5)\lambda_7 \end{bmatrix} \cdots$$

$$\cdots \begin{bmatrix} \lambda_1\lambda_3(\lambda_7+\lambda_8) & \lambda_6(\lambda_1+\lambda_3)(\lambda_4+\lambda_5) \\ -\lambda_6\lambda_3\lambda_7+(\lambda_1+\lambda_2+\lambda_6)\lambda_3(\lambda_7+\lambda_8) & \lambda_6\lambda_3\lambda_4 \\ (\lambda_1+\lambda_2+\lambda_6)(\lambda_1+\lambda_3)(\lambda_7+\lambda_8)-\lambda_6(\lambda_1+\lambda_3)\lambda_7 & \lambda_6(\lambda_1+\lambda_3)\lambda_4 \\ \lambda_1\lambda_3\lambda_7 & (\lambda_1+\lambda_2+\lambda_6)(\lambda_1+\lambda_3)(\lambda_4+\lambda_5)-\lambda_1\lambda_3\lambda_4 \end{bmatrix}$$

$$\det(M) = \frac{1}{(\lambda_1+\lambda_2+\lambda_6)(\lambda_1+\lambda_3)(\lambda_4+\lambda_5)(\lambda_7+\lambda_8)-\lambda_1\lambda_3\lambda_4(\lambda_7+\lambda_8)-\lambda_6(\lambda_1+\lambda_3)(\lambda_4+\lambda_5)\lambda_7}$$

For the calculation of MTTF value, the first row of the matrix N has to be summed up:

$$MTTF = \frac{1}{\det(M)}\cdot((\lambda_1+\lambda_3)(\lambda_4+\lambda_5)(\lambda_7+\lambda_8)+\lambda_1(\lambda_4+\lambda_5)(\lambda_7+\lambda_8)+ \tag{18}$$
$$+\lambda_1\lambda_3(\lambda_7+\lambda_8)+\lambda_6(\lambda_1+\lambda_3)(\lambda_4+\lambda_5))$$

The following values are assumed:

$$\lambda_1 = \frac{1}{31541}\cdot 10^{-7}\frac{1}{h} \tag{19}$$

$$\lambda_2 = \beta\cdot\lambda_1 = \left(\frac{0.005}{2}+\frac{0.02}{2}\right)\cdot 10^{-7}\frac{1}{h} \tag{20}$$

$$\lambda_3 = \frac{1}{\tau_{\mathrm{Re}\,pair}} = \frac{1}{\tau_{Diagnosis}} = \frac{1}{100}\cdot\frac{1}{3600000}\frac{1}{h} \tag{21}$$

$$\lambda_4 = \frac{1}{\tau_{\text{Re } juvenation}} = \frac{1}{300} \cdot \frac{1}{3600000} \frac{1}{h} \tag{22}$$

$$\lambda_5 = 10^{-7} \frac{1}{h} \tag{23}$$

$$\lambda_6 = \frac{1}{\tau_{\text{Re } juvenationCyclic}} = \frac{1}{0.25} \frac{1}{h} \tag{24}$$

$$\lambda_7 = \frac{1}{\tau_{\text{Re } juvenation}} = \frac{1}{300} \cdot \frac{1}{3600000} \frac{1}{h} \tag{25}$$

$$\lambda_8 = 10^{-7} \frac{1}{h} \tag{26}$$

According to the IEC 61508 part 6 [IEC00] a β-factor is used, to calculate the rate λ_2. This is the rate of a failure of the safety-related system due to a common cause failure. For the system with SES the given β-factor for diverse systems with good diagnostic tests has been choosen from IEC 61508 with the ratio of dangerous undetectable and dangerous detectable from [Bör04].

Consequently with equation 18 to 26 the MTTF could be calculated:

$$MTTF \approx \text{ 3,420,000 y} \tag{27}$$

6. Conclusion

In this paper a software rejuvenation model is proposed, in which SES is supplemented with the partial rejuvenation. Due to the SES more failures could be detected, so that it gets more likely, that the system is working in state 1. Nevertheless the software rejuvenation may not be successful under some circumstances. This leads to a failure of the system, but in total the reliability gets higher due to the fact that former undetected failures will be eliminated during the rejuvenation process.

As shown before with the help of the Markov modell of the two examplary systems it was possible to point out a great improvement of the system reliability by the calculation of a CTMC. Without coded processing and software rejuvenation the assumed system will have a MTTF of 2,170 years (cf. (13)). This means, that one system will fail every 2,170 years, which sounds quite good. But if you assume, that this system is running in one million cars, then about 460 cars will fail every year and this sounds no longer as good as before. With coded processing and with software rejuvenation the assumed system will have a MTTF of 3,420,000 years (cf. (27)). In the point of view like before,

this also means that only one car will fail every three to four years, if one million cars are running with the system safeguarded by SES in combination with software rejuvenation.

Of course the calculation of the MTTF could be done also for ASIL A and ASIL D systems. In Tab. 1 again the hardware failure rates defined in the ISO 26262 [ISO11] are used. It could be seen that the failure rate of the choosen hardware has a direct influence on the MTTF of the system regardless of whether SES is used or not.

	ASIL A	**ASIL B/C**	**ASIL D**
failure rate	10^{-6} h^{-1}	10^{-7} h^{-1}	10^{-8} h^{-1}
MTTF without SES	217 y	2,170 y	21,700 y
MTTF with SES	342,000 y	3,420,000 y	34,200,000 y

Table 1: **Comparisson of MTTF of ASIL A to D systems according ISO 26262.**

A further step would be to do a Semi-Markov calculation using the extended Weibull-function instead of the constant lambda values to be able to take the whole lifetime of an electronic system into account. With a Semi-Markov modell it would be even possible to calculate the optimal rejuvenation interval, which will improve the system's availability further.

Bibliography

[Bao05] Bao, Y.; Sun, X.; Trivedi, K.S.: A workload-based analysis of software aging and rejuvenation. IEEE Transactions on Reliability, Vol. 54, No. 3, pages 541-548, 2005

[Bör04] Börcsök, J.: Electronic Safety Systems: Hardware Concepts, Models, and Calculations. Hüthig, ISBN 978-3778529447, 2004

[Bra12] Braun, J.; Mottok, J.; Miedl, C.; Geyer, D.; Minas M.: Capability of single hardware channel for automotive safety applications according to ISO 26262. In Proceedings of the International Conference on Applied Electronics 2012 (AE2012) in Pilsen, September 2012.

[Cas01] Castelli, V.; Harper, R. E.; Heidelberger, P.; Hunter, S.W.; Trivedi, K.S.; Vaidyanathan, K.; Zeggert, W.P.: Proactive management of software aging. IBM Journal of Research & Development, Volume 45 Issue 2, pages 311-332, March 2001

[Cor11] Cotroneo, D.; Natella, R.; Pietrantuono, R.; Russo, S.: Software Aging and Rejuvenation: Where we are and where we are going. The 3rd International Workshop on Software Aging and Rejuvenation (WoSAR 2011), 2011

[Doh00n] Dohi, T.; Goseva-Popstojanova, K.; Trivedi, K.S.: Analysis of software cost models with rejuvenation. In Proceedings of the 5th IEEE International Symposium on High Assurance Systems Engineering, pages 25-34, Albuquerque, New Mexico, November 2000

[Doh00d] Dohi, T.; Goseva-Popstojanova, K.; Trivedi, K.S.: Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. In Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing (PRDC 2000), pages 77-84, Los Angeles, CA, December 2000

[For89] Forin, P.: Vital Coded Microprocessor Principles and Application for Various Transit Systems. IFAC Control, Computers, Communications, pages 79–84, Paris, 1989

[Gar95] Garg, S.; Puliafito, A.; Telek, M.; Trivedi, K.S.: Analysis of software rejuvenation using Markov regenerative stochastic Petri Nets. In Proceedings of the Sixth International Symposium on Software Reliability Engineering, pages 180-187, Toulouse, France, October 1995

[Gar98] Garg, S.; van Moorsel, A.; Vaidyanathan, K.; Trivedi, K.S.: A methodology for detection and estimation of software aging. In Proceeding of the International Symposium on Software Reliability Engineering (ISSRE 1998), pages 283-292, November 1998

[Hil07] von Hilgers, P.; Velminski, W.: Andrej A. Markov: Berechenbare Künste. Mathematik, Poesie, Moderne. Diaphanes, ISBN 978-3935300698, 2007

[Hua95] Huang, Y.; Kintala, C.; Kolettis, N.; Fulton, N.D.: Software Rejuvenation: Analysis, Module and Applications. In Proceedings of the 25th International Symposium on Fault Tolerant Computer Systems, pages 381-390, Pasadena, California, June 1995

[IEC00] CEI/IEC 61508-6:2000: Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 6: Guidelines on the application of lEG 61508-2 and lEC 61508-3

[ISO11] ISO 26262: Road vehicles - Functional safety, International Organization for Standardization, 2011

[Jin05] Jing, Y.; Jian, X.; Xue-lLong, Z.; Feng-Yu, L.; Modeling and availability analysis of nested software rejuvenation policy. In Proceedings of IEEE Intl. Conf. on Systems, Man and Cybernetics, Vol. 1, pages 34-38, 2005

[Kou05] Koutras, V.P.; Platis, A.N.: Optimizing the amount of the resources on a computer system using software rejuvenation. In Proceedings of European Safety and Reliability Conference, pages 1187-1192, 2005

[Kou10] Koutras, V.P.; Platis, A.N.: Semi-Markov performance modelling of a redundant system with partial, full and failed rejuvenation. International Journal of Critical Computer-Based Systems 2010, Vol.1 No.1/2/3, pages 59-85, 2010

[Li02] Li, L.; Vaidyanathan, K.; Trivedi, K.S.: An approach for estimation of software aging in a web server. In Proceedings of the 2002 International Symposium on Empirical Software Engineering, pages 91-102, Nara, Japan, October 2002

[Mot12] Mottok, J.; Schiller, F.; Zeitler, T.: Embedded Systems – Theory and Design Methodology. Chapter 2: Safely Embedded Software for State Machines in Automotive Applications, InTech, March 2012

[Par94] Parnas, D.L.: Software aging. In Proceedings of 16th International Conference on Software Engineering, pages 279-287, ACM Press, New York, 1994

[Sar09] Saravakos, P.K.; Gravvanis, G.A.; Koutras, V.P.; Platis, A.N.: A Comprehensive Approach to Software Aging and Rejuvenation on a Single Node Software System. Proceedings of The 9th Hellenic European Research on Computer Mathematics & its Applications Conference (HERCMA 2009), 2009

[Sch06] Schiller, F.; Mattes, T.: An Efficient Method to Evaluate CRC-Polynomials for Safety-Critical Industrial Communication. Journal Of Applied Computer Science, Vol. 14, No. 1, 2006

[She03] Shereshevsky, M.; Crowell, J.; Cukic, B.; Gandikota, V.; Liu, Y.: Software aging and multifractality of memory resources. In Proceedings of the 2003 International Conference on Dependable Systems and Networks, pages 721-730, San Francisco, June 2003

[Vai99] Vaidyanathan, K.; Trivedi, K.S.: A measurement-based model for estimation of resource exhaustion in operational software systems. In Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE 1999), pages 84-93, Boca Raton, Florida, November 1999

[Vai01] Vaidyanathan, K.; Harper, R.E.; Hunter, S.W.; Trivedi, K.S.: Analysis and implementation of software rejuvenation in cluster systems. In Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer System, pages 62-71, Cambridge, Massachusetts, USA, June 2001

[Xie05] Xie, W.; Yiguang, H.; Trivedi, K.S.: Analysis of a two-level software rejuvenation policy. Reliability Engineering and System Safety, Vol. 87, No. 1, pages 13-22, 2005

Modellbasierte Sicherheitsanalysen im BMBF-Förderprojekt e performance

Rasmus Adler[1], Sören Kemmann[1], Markus Schurius[2], Dr. Christian Allmann[3]

[1]Fraunhofer IESE, Abteilung ESQ, Fraunhofer-Platz 1, 67663 Kaiserslautern
[2]Audi Electronics Venture GmbH, I/AEV-23, Sachsstr. 18, 85080 Gaimersheim
[3]AUDI AG, I/PG-EE, 85045 Ingolstadt

Abstract: Der Artikel behandelt aktuelle Herausforderungen, die in dem Spannungsfeld von Funktionaler Sicherheit und der Integration von software-intensiven Fahrfunktionen bei verteilter Entwicklung entstehen. Er stellt Safety Engineering Methoden aus der Wissenschaft vor und beschreibt, wie diese Methoden erfolgreich in dem BMBF-Förderprojekt *e performance* angewendet wurden. In diesem Projekt entwickelt Audi mit mehreren Partnern aus Industrie und Forschung ein Elektrofahrzeug. Der Artikel schildert die zugrundeliegende Motivation, beschreibt das allgemeine Vorgehen und stellt erzielte Ergebnisse am Beispiel des elektrifizierten Antriebsstrangs vor.

1 Motivation

In der Vergangenheit wurde einer software-basierten Fahrzeugfunktionalität – wie beispielsweise der Längsdynamikregelung – jeweils eine dedizierte Rechnerplattform mit eigener Sensorik und Aktorik zugeordnet. Da diese Lösung aufgrund von Bauraum- und Energielimitierung die Grenzen der Skalierbarkeit erreicht hat, teilen sich bereits heute mehrere Software-Systeme die selben Ressourcen und können nur noch bedingt unabhängig voneinander entwickelt werden.

Um eine verteilte Entwicklung zwischen einem Automobilhersteller und seinen Lieferanten zu ermöglichen, müssen die Fahrzeugfunktionalitäten in interagierende Funktionen partitioniert werden. Diese Partitionierung wird durch die Funktionsarchitektur des Gesamtsystems beschrieben, die wiederum mit Sprachen wie SysML [We08] oder EAST-ADL [DST05] in Modellen dargestellt werden kann. Sie ermöglicht es, die komplexen Zusammenhänge von vernetzten Funktionen besser zu verstehen, dient als Kommunikationsmittel zwischen den beteiligten Parteien in der verteilten Entwicklung und schafft durch Informationsstrukturierung die Grundlage für die Gewährleistung von Funktionaler Sicherheit. Funktionale Sicherheit gemäß ISO 26262 [1] ist eine Systemeigenschaft, die garantiert, dass die Risiken sowohl durch Fehler bei der Umsetzung der spezifizierten Systemfunktionalitäten als auch durch zufällige HW-Fehler akzeptabel sind.

Safety Engineering und die Entwicklung der Funktionsarchitektur stehen in einer engen Beziehung zueinander: Die Funktionsarchitektur ist die Basis für die Gewährleistung der Funktionalen Sicherheit. Jede Änderung an der Architektur hat potentiell Einfluss auf die Funktionale Sicherheit. Andererseits haben Entwicklungsartefakte, die im Sicherheitslebenszyklus entstehen direkt Auswirkungen auf die Architektur.

Fehlererkennungs- und Fehlerbehandlungsmechanismen führen beispielsweise zu neuen Funktionen, die ebenfalls in der Architektur abgebildet und analysiert werden. Aufgrund der steigenden Komplexität von Funktionsarchitekturen wird es daher immer aufwändiger, die Konsistenz zwischen der Funktionsarchitektur und den Safety-Artefakten in Form von Sicherheitsanalysen und -konzepten zu garantieren. Herkömmliche Safety Engineering Methoden aus der Praxis stoßen an ihre Grenzen. Betrachtet man beispielsweise eine traditionelle Fehlerbaumanalyse (engl. Fault Tree Analysis, FTA), dann bezieht sich der resultierende Fehlerbaum auf einen bestimmten Stand der Funktionsarchitektur. Änderungen an der Funktionsarchitektur ziehen oftmals eine entsprechende Modifikation im Fehlerbaum nach sich. Dies ist sehr aufwendig da es im Allgemeinen nicht klar ist welche Teile des Fehlerbaums wie angepasst werden müssen. Die dazu benötigte Änderungseinflussanalyse, wird zudem durch eine unvollständige Werkzeugintegration zwischen den Disziplinen System bzw. Software Engineering und konventionellem Safety Engineering erschwert.
Die gleiche Problematik betrifft auch das funktionales Sicherheitskonzept, solange es wie bisher weitestgehend textbasiert erstellt wird und grafische Notationen wie beispielsweise Fehlerbäume oder Modellteile anderer Entwicklungswerkzeuge lediglich als Abbildungen zum besseren Verständnis inkludiert. Das funktionale Sicherheitskonzept legt Sicherheitsanforderungen für die Funktionen in der Funktionsarchitektur fest und erklärt warum die Sicherheitsanforderungen ausreichen um die Sicherheitsziele zu erreichen. Wenn die Funktionsarchitektur geändert werden muss, ist somit die Anpassung des Sicherheitskonzepts ebenfalls sehr zeitaufwendig.
Um diese Konsistenzprobleme zu lösen muss eine verbesserte Verfolgbarkeit zwischen der funktionalen Architektur, den Fehlerbäumen und dem funktionalen Sicherheitskonzept hergestellt werden. Dazu wird ein Ansatz benötigt bei dem alle sicherheitsrelevanten Informationen die im Projektverlauf entstehen in einem gemeinsamen Modell integriert werden können. Bisher sind diese noch häufig in separaten Entwicklungsartefakten abgebildet, die manuell verlinkt werden müssen. Ausgangspunkt ist dabei die modellbasierte Entwicklung, beziehungsweise die systematische Erfassung von Funktionen und deren Abhängigkeiten anhand logischer Schnittstellen in einem Modell. Mit der bruchfreien Integration von Sicherheitsanalysen in die modellbasierte Entwicklung wird eine explizite und durchgängige Sicherheitsargumentation ermöglicht. Desweiteren vereinfacht diese modellbasierte, iterative Vorgehensweise eine kohärente Entwicklung von teilautomatisiert erstellten Sicherheitskonzepten und modularen Architekturen. Zudem wird durch die Ausdrucksmächtigkeit von Modellierungssprachen, z.B. in Form von unterschiedlichen Diagrammtypen, die Sicherheitsfunktion exakter spezifiziert und kann in definierbaren Sichten auf das Modell leichter verstanden und weiterentwickelt werden. Damit steigt auch die Wiederverwendbarkeit von Entwicklungsartefakten.
Da dieses Verfahren auf der Analyse von Funktionslogik basiert ist es für die konventionellen, d.h. im Fahrzeug verteilten Funktionen in gleichem Maße geeignet wie für hochintegrierte Funktionen auf einem einzigen Steuergerät. Speziell für diesen zweiten Fall bildet es vielmehr die Grundlage um Verteilungsentscheidungen (HW/SW-Deployment) in Bezug auf das Ausführungsverhalten so zu spezifizieren dass es weder zu Verletzungen von Echtzeitbedingungen noch von Sicherheitseigenschaften führt. Um z.B. die Einhaltung von Ende-zu-Ende Fehlertoleranzzeiten einer Funktion zu untersuchen kann ein Timing-Modell wie in [Sc12] beschrieben in das hier vorgestellte,

modellbasierte Sicherheitsanalyse-Framework integriert werden. Ziel ist dabei die iterative Entwicklung von Lösungen, die unterschiedliche Aspekte berücksichtigen und eine kontinuierliche Optimierung der Funktions- und Steuergerätearchitektur erleichtern.

Der aktuelle Stand der Technik im Bereich sicherheitsgerichteter Entwicklung besitzt die folgenden Verbesserungspotentiale:

- Vermeidung von Brüchen in der Werkzeugkette (Sicherheitsanalyse vs. Modellierung) zur Verbesserung der Durchgängigkeit von sicherheitsrelevanten Informationen im Sicherheitslebenszyklus bzw. entlang des V-Modells
- Überwindung von organisatorischen Grenzen mittels Modularisierung und Integration von sicherheitsrelevanten Informationen in einem gemeinsamen Modell, sowohl hinsichtlich unterschiedlicher Rollen und Disziplinen innerhalb einer Firma als auch über die Schnittstelle zwischen OEM und Lieferanten hinweg
- Unterstützung einer effizienten und inkrementellen Bewertung des Sicherheitsnachweises

Die Akzeptanz der Anwender für eine intensivere, in diesem Falle phasen- und disziplinübergreifende Entwicklung auf einem gemeinsamen Informationsmodell ist vorhanden.

Dieser Artikel zeigt, wie innovative Safety Engineering Methoden genutzt werden können, um mit der steigenden Komplexität bei der Integration von Fahrzeugfunktionalitäten umzugehen. Er beschreibt zunächst einen modellbasierten Lösungsansatz, um die Komplexität software-intensiver, vernetzter Fahrfunktionen zu beherrschen. Anschließend wird das BMBF-Förderprojekt *e performance* vorgestellt und erklärt, wie darin Sicherheitskonzepte modellbasiert entwickelt wurden.

2 Lösungsansatz

Der Wunsch nach mehr Kundennutzen ist die Ursache für die steigende Anzahl und die Vernetzung von Fahrzeugfunktionen – eine Charakteristik, die speziell auf Fahrerassistenzsysteme zutrifft. Ein Mittel, um diese Komplexität zu beherrschen ist die Verwendung von modellbasierten Entwicklungsmethoden. Komplexitätsreduktion durch Abstraktion hilft systematische Fehler zu vermeiden und begünstigt somit die Funktionale Sicherheit. Aufgrund einer verbesserten Formalisierung der zu entwickelnden Funktionen wird modellbasierte Entwicklung von modernen Sicherheitsnormen wie der ISO 26262 empfohlen.
Um mit der steigenden Komplexität der verteilten Entwicklung von integrierten Fahrzeugfunktionen umzugehen, kann ein Funktionsmodell erstellt werden, das beschreibt, wie die Fahrzeugfunktionalitäten durch eine Komposition von Teilfunktionen realisiert werden. Modelle des Safety Engineering ermöglichen es hingegen mit den komplexen Zusammenhängen von sicherheitsrelevanten Aspekten umzugehen. Ein wesentlicher Punkt, der beim Safety Engineering berücksichtigt werden muss, ist die Ursache-Wirkungsbeziehung von Fehlern. In diesem Kontext muss insbesondere

analysiert werden, wie sich Fehler von Funktionen auf Fahrzeugfunktionalitäten auswirken. Hierfür werden häufig Fehlerbäume verwendet, in denen für jede zuvor ermittelte Gefährdung (in absteigender Reihenfolge) eine Baumstruktur entwickelt wird – bis schließlich alle möglichen Ursachen, die zum Eintreten der Gefährdung führen können, identifiziert sind. Die Baumzweige wiederum sind über logische Operatoren miteinander verknüpft. Durch Auswerten der Minimalschnitte können Ereigniskombinationen ermittelt werden, die im Falle ihres gleichzeitigen Eintretens zur Gefährdung bzw. zu einer Verletzung des Sicherheitsziels führen können.
Fehlerbäume und andere Safety-Modelle beziehen sich häufig auf das Funktionsmodell. Dieses spezifiziert die zu entwickelnden Funktionen und deren Zusammenhänge. Modelle für die Analyse von Fehlern in Funktionen untersuchen zum Beispiel fehlerhafte Abweichungen von Funktionsspezifikationen und deren Auswirkungen. Andere Safety-Modelle beschreiben funktionale Sicherheitsanforderungen bzw. sicherheitsrelevante Teile der Funktionsspezifikation. Auch wenn sich Safety-Modelle auf die Information im Funktionsmodell beziehen, werden sie traditionell unabhängig vom Funktionsmodell und mit separaten Werkzeugen erstellt. Gemeinsamkeiten werden somit redundant und möglicherweise inkonsistent zueinander modelliert.
Um dieses Defizit im Hinblick auf Fehlerbaumanalysen zu beseitigen, wurde in [DT08] ein Ansatz entwickelt wie Fehlerbäume in ein Funktionsmodell integriert werden können. Darin besitzt jede Komponente in Funktionsmodell einen eigenen Fehlerbaum. Dies ist nicht nur im Rahmen der verteilten Entwicklung ein großer Vorteil, denn Konsistenzprobleme zwischen Fehlerbaum und Funktionsmodell treten auch bei einer nicht verteilten Entwicklung auf. Um die Konsistenzproblematik auch bei der Sicherheitskonzeptentwicklung in den Griff zu bekommen wurde ein ähnlicher Ansatz in [Do09] entwickelt. In dem Ansatz hat jede Komponente in dem Funktionsmodell ein eigenes Sicherheitskonzept. Um die beiden Integrationsansätze zu kombinieren und noch weitere Safety-Modelle integrieren zu können wurde in [Ad11] ein Ansatz entwickelt, der es ermöglicht verschiedene Safety-Sichten für ein Funktionsmodell zu erstellen. Gemäß des Prinzips *Separation of Concerns* können die Sichten separat modelliert und evaluiert werden.
Der Stand der Technik bietet viele verschiedene Arten von Safety-Modellen, die man als Safety- Sichten in ein Funktionsmodell integrieren könnte. Die Modelle kann man gemäß ihrem Zweck als analytisch oder als konstruktiv klassifizieren. Analytische Modelle dienen der Ursachenanalyse. Gemäß Fenelon et al. [Fe94], kann man analytische Ansätze anhand ihrer Suchrichtung nach Ursachen (von Gefährdungen) charakterisieren. Hierbei werden explorative, induktive, deduktive und beschreibende Ansätze unterschieden. Um den Aspekt Software und Funktion adäquater zu adressieren, haben wir dieses Schema etwas angepasst und erweitert. So unterscheiden wir grob, HAZOP (engl. Hazard and Operability) Analysen [Pu99][Ch93][RCC99], reine Vorwärtsanalyseansätze wie FMEA (engl. Failure Mode and Effects Analysis), reine Rückwärtsanalyseansätze wie FTA, FMEA-FTA kombinierte bidirektionale Analyseansätze [Pa04][LW99][PD99], Inspektions-ähnliche sowie formale Ansätze [De09]. Die Anzahl der analytischen Modelle zur Ursachenidentifikation ist deutlich größer als die Anzahl der konstruktiven Safety-Modelle zur Beschreibung der Maßnahmen gegen identifizierte Ursachen und Gefährdungen. Der populärste konstruktive Ansatz zur Darstellung des Sicherheitsnachweises (engl. Safety Case) ist die Goal Structuring Notation (GSN) [Wi96]. Die GSN eignet sich auch bedingt zur

Modellierung von Sicherheitskonzepten. Der Sicherheitsnachweis nach ISO 26262 ist das finale Arbeitsergebnis im Sicherheitslebenszyklus und beinhaltet die Gesamtheit der Entwicklungsartefakte, die zur Beurteilung und zum Nachweis der Funktionalen Sicherheit herangezogen werden. Die wichtigsten Kriterien für den Sicherheitsnachweis sind Vollständigkeit, Widerspruchsfreiheit und Durchgängigkeit, wobei speziell im Automotivebereich zusätzlich Vergleichbarkeit sowie Modularität zunehmend an Bedeutung gewinnen. Der Sicherheitsnachweis wird jeweils für eine Betrachtungseinheit erstellt, beispielsweise für eine sicherheitsrelevante Fahrzeugfunktion, die in einem System oder über mehrere Systeme verteilt realisiert ist. Er muss eine für den Sicherheitsverantwortlichen nachvollziehbare und durchgängige Sicherheitsargumentation dokumentieren. Diese Argumentation basiert auf Evidenzen in Form von Entwicklungsartefakten und zeigt auf, woher Sicherheitsanforderungen stammen, wie diese sowohl konstruktiv als auch prozessual korrekt implementiert wurden und dass deren Wirksamkeit auf Fahrzeugebene nachgewiesen wurde.
Der Hauptnutznießer eines klar strukturierten und durchgängigen Sicherheitsnachweises ist der Sicherheitsverantwortliche einer Fahrfunktion bzw. eines Systems. Entwicklungsingenieure können den Sicherheitsverantwortliche mit einer großen Menge sorgfältig erstellter Materialien wie FTAs oder FMEAs versehen, aber isoliert und ohne strukturierte Sicherheitsargumentation reicht dies nicht aus, um einen systematischen Nachweis der Sicherheit eines Systems zu ermöglichen. Der natürliche Zeitpunkt zur Erstellung des Sicherheitsnachweises liegt demnach nach der Beendigung der eigentlichen Produktenwicklung. Da aber speziell in der Automotiv Domäne die komplette Entwicklung meist iterativ und inkrementell erfolgt, muss auch der Sicherheitsnachweis aus Effizienzgründen inkrementell erstellt und am Ende der Produktentwicklung auf Vollständigkeit und Korrektheit überprüft werden können.
Für eine entwicklungsbegleitende Erstellung des Sicherheitsnachweises haben Graydon und Knight [GKS07] mit dem Assurance Based Development (ABD) ein Konzept vorgestellt, welches die frühzeitige und schrittweise Erstellung des Sicherheitsnachweises ermöglicht. Im Rahmen seiner Vorstellung der ABD nutzt Knight die GSN als Modellierungssprache für Sicherheitsnachweise. Egal wie der Sicherheitsnachweis dargestellt wird, er wird immer Sicherheitsanforderungen beinhalten. Diese sind bei hinreichend komplexen Systemen in der Regel nicht direkt aus Gefährdungen oder Sicherheitszielen auf Fahrzeugebene ableitbar sondern werden in Form von Sicherheitskonzepten unter Berücksichtigung von Funktionsarchitektur und anhand von Sicherheitsanalysen entwickelt.
Die einzige Modellierungssprache, die eigens für die Modellierung von Sicherheitskonzepten entwickelt wurde geht auf [Do09] zurück. Sie wurde am Fraunhofer IESE entwickelt, um den speziellen Herausforderungen der Erstellung eines Sicherheitskonzepts für komplexe, verteilt entwickelte Systeme zu begegnen. Diese Modellierungssprache für Sicherheitskonzepte beinhaltet insbesondere intuitive und hinreichend formale Konzepte zur Modularisierung, sowie die Unterstützung der Modellierung von ASIL[1]- Dekomposition. Durch ASIL-Dekomposition ist es möglich,

[1] ASIL (engl. Automotive Safety Integrity Level): Eine von vier Kategorien (A-D) zur Spezifizierung der notwendigen Anforderungen und Methoden gemäß ISO 26262 für die Betrachtungseinheit um ein akzeptables Restrisiko zu erreichen. ASIL D stellt dabei die höchsten Anforderungen dar.

Sicherheitsanforderungen redundant auf mehrere unabhängige Architekturelemente abzubilden und diesen einen reduzierten ASIL zuzuordnen.
Die Integration von Safety-Modellen in das Funktionsmodell wurde bereits an vielen akademischen und industriellen Beispielen erfolgreich evaluiert [Ad12]. In Bezug auf die Automotive Domäne, war die Komplexität der Beispiele allerdings nicht vergleichbar mit derer von heutigen Fahrerassistenzsystemen.

3 BMBF-Förderprojekt e performance

Das BMBF-Förderprojekt *e performance* [2] war prädestiniert, um die Praxistauglichkeit der Safety-Sichten zu evaluieren. Ziel dieses Projekts war die ganzheitliche Entwicklung eines Elektrofahrzeugs, wobei die entstehenden Komponenten in unterschiedlichen Fahrzeugkonzepten möglichst modular verwendbar sein mussten.
Im Förderprojekt hatte das Arbeitspaket „Funktionale Sicherheit“ die Aufgabe, ein ganzheitliches Sicherheitskonzept auf Gesamtfahrzeugebene für elektrisch betriebene Fahrzeuge zu erstellen. Im Fokus der Betrachtung standen dabei übergeordnete, vernetzte Funktionen, die sich aus einzelnen Subsystemen oder Komponenten heraus nicht effizient absichern ließen. Beispiele hierfür sind – wie in Abbildung 1 dargestellt – unter anderem die Betriebsstrategie, welche die Koordination zwischen Antrieb, Bremse, Fahrer und Fahrerassistenzsystemen darstellt, der Ladevorgang des Fahrzeugs sowie assoziierte Querschnittsfunktionen, wie das Energiemanagement.

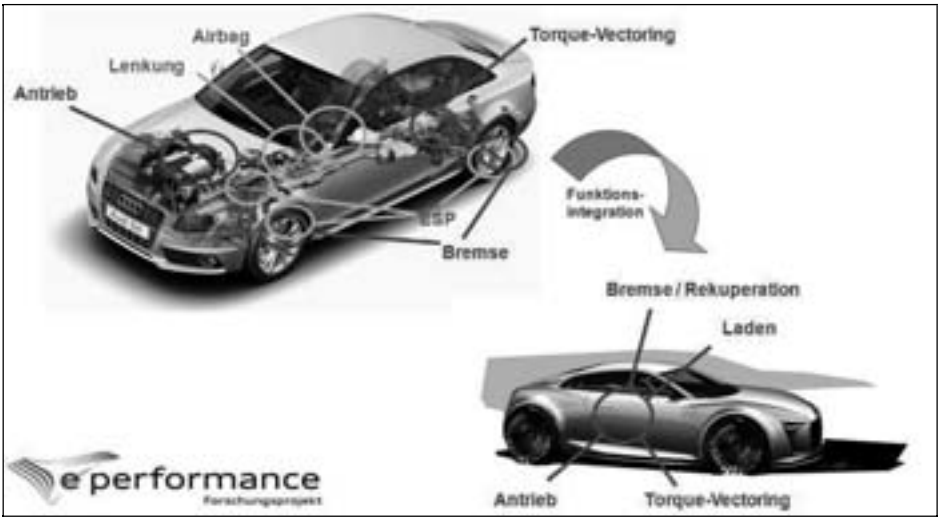


Abbildung 1: Hochintegration von Fahrfunktionen

Neue Funktionen, die Integration und Erweiterung bestehender Umfänge und prinzipielle Veränderungen im System und an den Systemgrenzen, ergeben insbesondere bei rein elektrisch betriebenen Fahrzeugen neue, für OEMs bisher weitgehend unbekannte Gefahrenquellen. Die ganzheitliche Analyse der Gefahren und Risiken auf Gesamtfahrzeugebene ist aufgrund der Komplexität und des Umfangs im Gegensatz zur Betrachtung eines einzelnen Subsystems jedoch ungleich anspruchsvoller. Dies gilt

insbesondere für die Entwicklung von vernetzten Funktionen mit stark verteilten Umfängen, wie es im vorliegenden Projekt der Fall ist. Um auch in diesem heterogenen Umfeld eine effiziente Sicherheitsanalyse zu gewährleisten, muss diese durch Modelle des Systems bzw. relevanten Teilsystemen und deren Fehlverhalten unterstützt werden.

4 Modellbasierte Sicherheitsanalysen und Sicherheitskonzepte

Das Fraunhofer IESE brachte seine Expertise im Bereich modellbasiertes Safety Engineering in das Förderprojekt ein. Es zeigte, wie Safety-Sichten auf das Funktionsmodell genutzt werden können, um mit den neuen Herausforderungen durch zunehmende Hochintegration von Fahrzeugfunktionen umzugehen.
Im Folgenden wird zunächst die allgemeine Vorgehensweise im Safety Engineering für Funktionale Sicherheit beschrieben. Anschließend wird erklärt welche Safety-Sichten in *e performance* evaluiert wurden.

4.1 Safety Engineering

Die sicherheitsgerichtete Entwicklung beginnt mit der Identifikation von Risiken durch Abweichungen vom Sollverhalten des Systems. Gemäß der ISO 26262 wurden dazu Gefährdungssituationen identifiziert und bezüglich ihres Risikos bewertet. In diesem Fall wurde für 5.600 Gefährdungssituationen bestimmt, wie hoch die notwendige Risikoreduktion sein muss, um Funktionale Sicherheit zu garantieren. Auf Basis dessen wurden anschließend die Sicherheitsziele abgeleitet. Ein solches bestand zum Beispiel darin abzusichern, dass das Differenzenmoment der Hinterräder den bestimmten Grenzwert nicht überschreitet, ab dem es zum Kontrollverlust über das Fahrzeug kommen kann. Die Sicherheitsmaßnahme für das genannte Sicherheitsziel erforderte unter anderem das Differenzenmoment der Hinterräder zu überwachen und diese in den momentenfreien Zustand zu versetzen, wenn das Differenzenmoment zu groß wird.

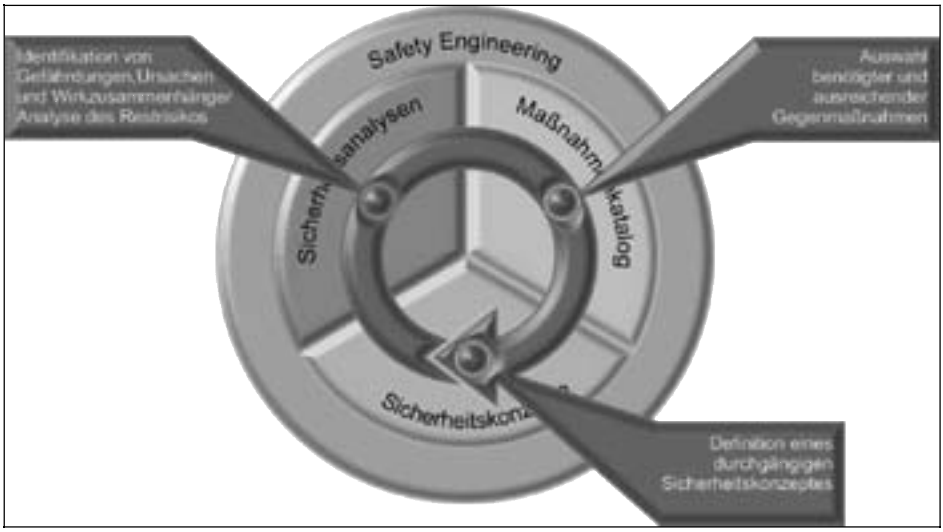


Abbildung 2: Iteratives Safety Engineering

Wie in Abbildung 2 veranschaulicht, ist die Maßnahmenidentifikation im Allgemeinen ein iterativer Prozess, bei dem analysiert wird, wie es dazu kommen kann, dass ein Sicherheitsziel verletzt ist. Sicherheitsanalysen werden dazu verwendet, um die Ursachen zu finden, warum ein oder mehrere Sicherheitsziele verletzt werden können. Basierend auf den identifizierten Ursachen werden im zweiten Schritt konstruktive Sicherheitsmaßnahmen systematisch ausgewählt.
Der iterative Prozess zwischen Sicherheitsanalysen und Maßnahmenauswahl endet sobald die Sicherheitsanalysen zeigen, dass alle Ursachen ausreichend durch Maßnahmen abgesichert sind. Um eine durchgängige, widerspruchsfreie und nachvollziehbare Darstellung der notwendigen Risikoreduktionen zu erhalten, wird in einem Sicherheitskonzept dokumentiert, welche Maßnahme warum ausgewählt wurde. Des Weiteren muss im Sicherheitskonzept auch dokumentiert sein, wie die Maßnahmen durch Sicherheitsfunktionen implementiert werden sowie welche Sicherheitsanforderungen an die normalem Systemfunktionen gestellt werden, um die Sicherheitsziele zu garantieren.

Um den Aufwand für das Safety Engineering zu minimieren wurden am Fraunhofer IESE Safety-Sichten für ein Funktionsmodell entwickelt. Abbildung 3 zeigt dieses Funktionsmodell[2], das im Rahmen des Projektes *e performance* erstellt wurde. Es beschreibt die funktionalen Zusammenhänge zwischen dem elektrifizierten Antriebsstrang und der Stromversorgung. Die Funktionen sind hierarchisch gruppiert. So sind beispielsweise die Funktionen für den Inverter, der den hinteren linken Elektromotor ansteuert, zu einer hierarchischen Funktion „AntriebsumrichterHL“ zusammengefasst. Für die Einregelung des Sollmoments ist eine Teilfunktion dieses Inverters verantwortlich.

[2] Hinweis: Die Details der abgebildeten Modelle sind für das weitere Verständnis unerheblich.

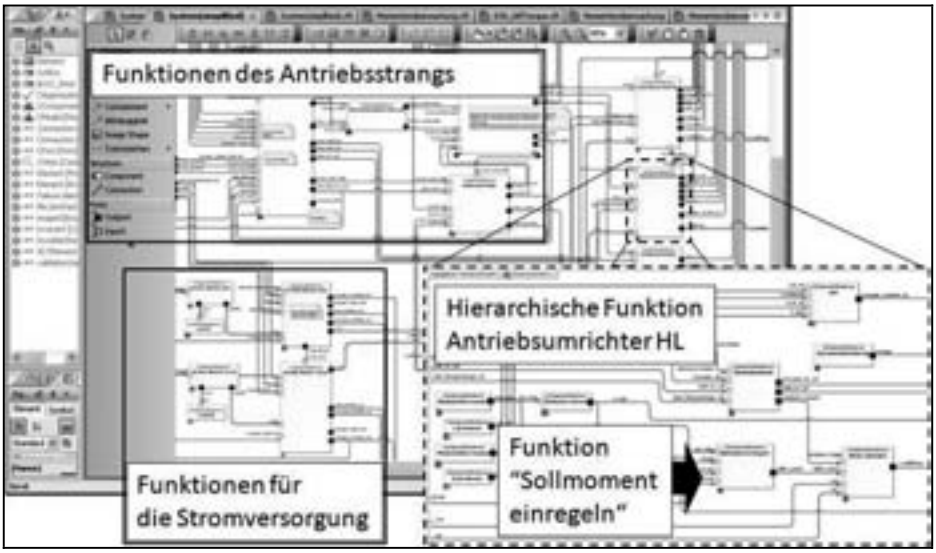


Abbildung 3: Funktionsmodell elektrifizierter Antriebsstrang im Projekt e performance

Im Folgenden wird beschrieben, mit welchen Sichten das Funktionsmodell erweitert wurde, um das Safety Engineering zu unterstützen.

4.2 Failure View

Eine wesentliche Sicht, um das Safety Engineering zu unterstützen, ist der *Failure View*. Dieser beschreibt die Ursache-Wirkungszusammenhänge von Fehlern und ermöglicht somit automatisierte Sicherheitsanalysen. Abbildung 4 zeigt den Failure View der Funktion „Sollmoment einregeln“. Er zeigt, wie Fehler von gesendeten Informationen bzw. Signalen mit internen Fehlern der Funktion und mit Fehlern von empfangenen Informationen bzw. Signalen zusammenhängen.
Die Fehler eines Ausgangsignals werden als „Fehlermodi“ bezeichnet und durch schwarze Dreiecke symbolisiert. Die Fehler eines Eingangssignals hingegen werden durch gelbe Dreiecke, interne Fehler wie bei der Fehlerbaumnotation durch Kreise dargestellt. Der Zusammenhang zwischen den Fehlern wird ebenfalls mit der üblichen Fehlerbaumnotation in Form von Booleschen Gattern modelliert. Um graphisch darzustellen, welcher Fehlermodus zu welchem Signal gehört, wird jeder Fehlermodus mit dem Port verbunden, über den das fehlerhafte Signal empfangen bzw. gesendet wird.

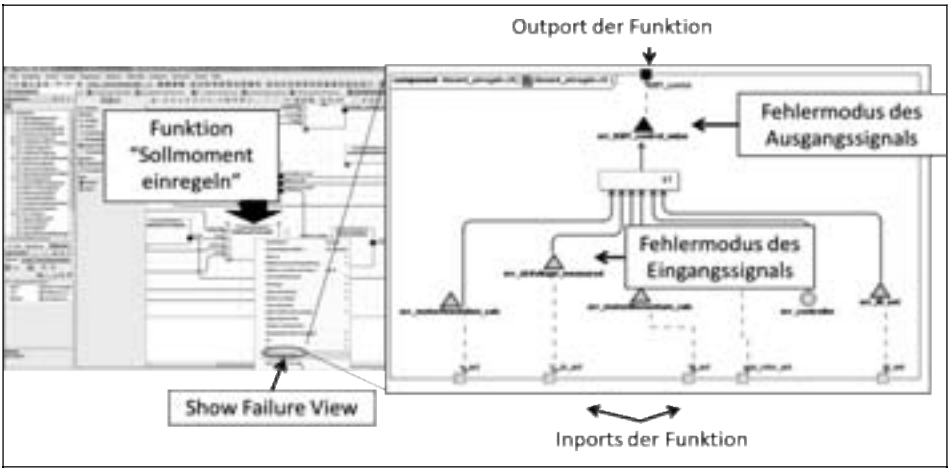


Abbildung 4: Failure View Sollmoment einregeln

Die obige Sicht zeigt jedoch nur, wie sich Fehler innerhalb einer Funktion fortpflanzen. Um zu beschreiben, wie sich Fehler von einer Funktion zur nächsten propagieren, wird für jede Verbindung im Funktionsmodell festgelegt, wie die Ausgangsfehlermodi der sendenden Funktion mit den Eingangsfehlermodi der empfangenen Funktion zusammenhängen. Abbildung 5 ist zu entnehmen, wie die funktionsübergreifende Fehlerfortpflanzung im Tool umgesetzt ist. Hierbei können verschiedene Failure Port Mappings für eine Verbindung festgelegt werden. Jedes dieser definiert, wie ein Ausgangsfehlermodus der sendenden Funktion einen Eingangsfehlermodus der empfangenen Funktion verursacht.

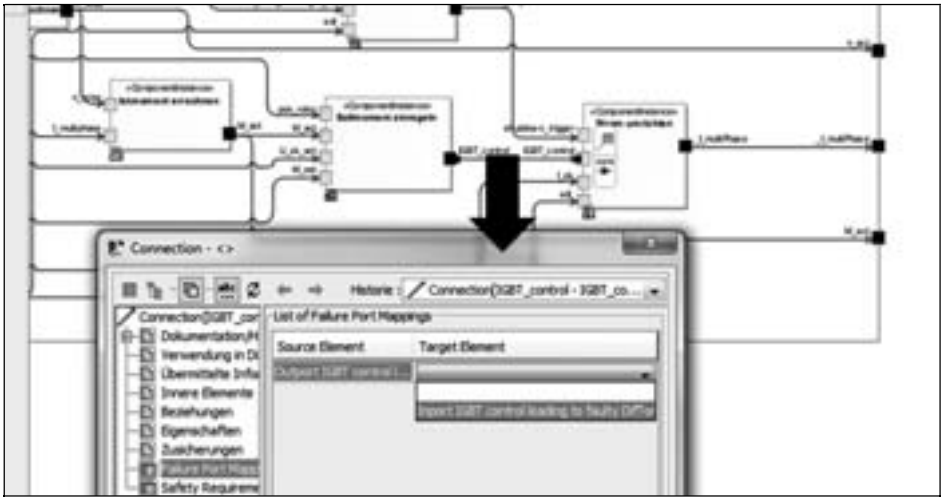


Abbildung 5: Fehlerfortpflanzung über Schnittstellen

Basierend auf den Failure Port Mappings werden die Failure Views der hierarchischen Funktionen generiert. Abbildung 6 verdeutlicht den generierten Failure View der

hierarchischen Funktion „AntriebsumrichterHL“. Er enthält alle Failure Views von den Unterfunktionen und beschreibt wie diese zusammenhängen. Im Beispiel enthält er den Failure View der Funktion „Sollmoment einregeln“ – als eine Unterfunktion der Funktion „AntriebsumrichterHL“. Der Failure View auf oberster Hierarchieebene beschreibt alle Fehlerzusammenhänge und somit alle notwendigen Informationen, um mit Hilfe von automatisierten Analysen herauszufinden, welche Fehlerkombinationen welche Sicherheitsziele verletzen.

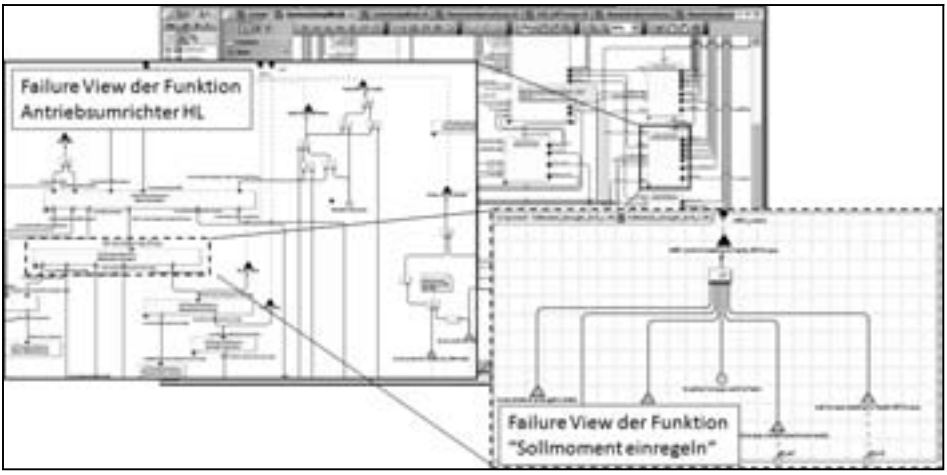


Abbildung 6: Hierarchischer Failure View

4.3 Safety Concept View

Eine weitere Sicht um das Safety Engineering zu unterstützen, ist der *Safety Concept View*. Dieser dokumentiert, was eine Funktion für ihre Ausgangssignale garantiert und wie diese Garantien mit Sicherheitsanforderungen der Funktion und den empfangenen Informationen zusammenhängen. Abbildung 7 zeigt den Safety Concept View der Funktion, die für die Erkennung eines zu hohen Differenzenmomentes verantwortlich ist. Die Funktion garantiert, dass sie ein Abschaltsignal sendet, wenn das hintere Differenzenmoment größer ist als ein definierter Grenzwert, der von den fahrdynamischen Eigenschaften des Fahrzeugs abhängt.
Um diese Garantie zu geben, fordert sie, dass die empfangenen Informationen über die Momente der hinteren Räder hinreichend genau sind. Des Weiteren implementiert sie die Sicherheitsanforderung, dass immer genau dann ein Abschaltsignal generiert wird, wenn die Differenz der empfangenen Radmomente den Grenzwert überschreitet.

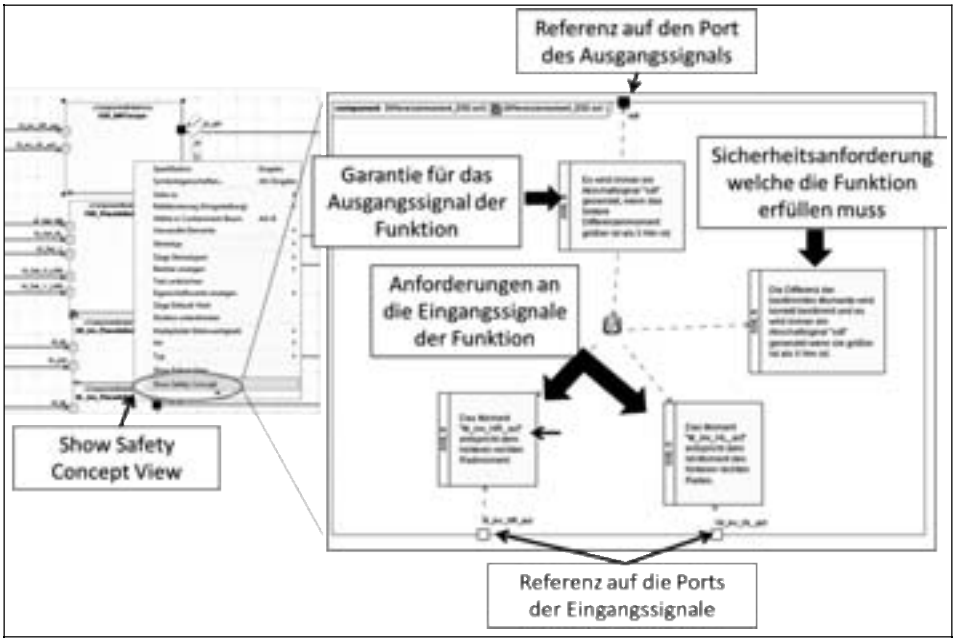


Abbildung 7: Safety Concept View

Der Safety Concept View lässt sich ebenso wie der Failure View hierarchisch im Modell in sogenannte Safety Concept Parts gliedern und unterstützt damit die iterativ-inkrementelle Entwicklung eines systematisch strukturierten und modularen Sicherheitskonzepts. Wir unterscheiden an dieser Stelle nicht zwischen einer Notation für funktionale oder technische Sicherheitskonzepte. Selbstverständlich ist es aber möglich in hierarchisch strukturierten Modellen funktionale Aspekte auf Ebene der logischen Funktionsarchitektur darzustellen, ebenso wie deren Verfeinerung durch Abbildung auf die technische Systemarchitektur, z.B. in UML Deployment-Diagrammen. Der zusicherungsbasierte Ansatz erleichtert die Wiederverwendung von Komponenten und die konsistente Einbindung von kontextfreien Sicherheitselementen (vgl. ISO 26262, Safety Element out of Context, SEooC). Solange an den Schnittstellen die Sicherheitsanforderungen der einen, durch ausreichende Sicherheitsgarantien der anderen Komponente erfüllt sind, besitzt auch die Komposition aus modularen Einzelkomponenten eine entsprechende Sicherheitsintegrität. Mit dieser Methode ist es nun möglich, aus verschiedenen kontextfreien Sicherheitselementen für bestimmte generische Sicherheitsfunktionen durch Komposition auf höherer Ebene eine ebenfalls kontextfreie und in diesem Sinne modulare Sicherheitsarchitektur, sozusagen eine Safety Architecture out of Context (SAooC), zu entwerfen.

5 Zusammenfassung und Ausblick

Allgemein ist der Failure View und seine Integration in einen modellbasierten Top-Down Entwicklungsprozess ein wesentlicher Schritt hin zu modularen Sicherheitsnachweisen, die es ermöglichen, Teile von Entwicklungsartefakten ohne

Mehraufwand wiederzuverwenden. Der Safety Concept View erlaubt es eine durchgängige Sicherheitsargumentation passend zu einer funktionalen Architektur zu modularisieren.
Im Projekt *e performance* wurden der Failure View und der Safety Concept View erfolgreich eingesetzt, um ein Sicherheitskonzept für den elektrifizierten Antriebsstrang zu entwickeln und modular zu beschreiben. Das Sicherheitskonzept wurde auf mehreren Ebenen verteilt implementiert. Die Antriebsumrichter erkennen und behandeln lokale Fehler eigensicher unter Verwendung von diversitären Prinzipien sowie deren Umsetzung auf dissimilarer Hardware. Im integrierten Antriebsstrang, bestehend aus mehreren Einzelantrieben, plausibilisiert sich die Antriebslogik gegenseitig. Ein separates Steuergerät überwacht die Längs- und Querdynamik auf oberster Systemebene und kann abhängig vom Fehlergrad den Antriebsstrang redundant momentenfrei schalten.
Im modellbasierten Entwicklungsprozess des elektrifizierten Antriebsstrangs erwies sich das Funktionsmodell als unumgänglich um die funktionalen Zusammenhänge zu verstehen. Eine informale Beschreibung basierend auf Textdokumenten (Item Definitions) wäre bei der stark verteilten Entwicklung nicht ausreichend gewesen, um mit der hohen Änderungsrate von neuen Funktionen umzugehen. Die beiden Sichten Failure View und Safety Concept View waren essentiell um eine Konsistenz zwischen dem Safety Engineering und der Entwicklung der Funktionsarchitektur zu gewährleisten. Insbesondere Änderungen der Funktionsarchitektur konnten effizient im Safety Engineering behandelt werden, indem nur der Failure View und der Safety Concept View von den betroffenen Funktionen angepasst wurden.
Das vom Fraunhofer IESE entwickelte Werkzeug für die Erstellung modellintegrierter Sicherheitsanalysen und Sicherheitskonzepte basiert auf XML und kann deshalb mit vertretbarem Aufwand in gängige Modellierungswerkzeuge integriert werden.

Systematische Wiederverwendung sowie synergetische Funktions- und Software-Baukästen sind wesentliche Bestandteile der Technologie-Roadmap von Audi. Die im Förderprojekt evaluierten Methoden und Werkzeuge für modellbasierte Sicherheitsanalysen und Sicherheitskonzepte sind vielversprechend und werden in das etablierte Vorgehen sukzessive integriert und erweitert. Aus diesem Grund schließt die Audi Electronics Venture GmbH Folgeaktivitäten an und beteiligt sich im BMBF-Förderprojekt SPES_XT [3] als Partner sowie auch auf europäischer Ebene im ITEA-2 Projekt SAFE [4] als OEM Advisor.
Mögliche Erweiterungen zum bisher Erreichten sind beispielsweise eine modellbasierte Kopplung von Item Definition mit datenbankgestützter Gefahren- und Risikoanalyse. Dabei ist es denkbar, die Aspekte der Gebrauchssicherheit in das gemeinsame Funktionsmodell einzubeziehen. Ein naheliegender Ansatz wäre die Verwendung von Anwendungsfall-, Aktivitäts- und Kontextdiagrammen, um damit die Interaktion des Fahrers mit der als funktionssicher angenommenen Fahrfunktion und der Systemumgebung abzubilden. Eine zusätzliche Anforderung ist sicherlich die Anbindung von Fahrsimulationen, um aus modellbasierten Szenarien abgeleitete Sicherheitskonzepte frühzeitig validieren zu können.

6 Literatur

[Ad11] Rasmus Adler, Dominik Domis, Kai Höfig, Sören Kemmann, Thomas Kuhn, Jean-Pascal Schwinn and Mario Trapp. Integration of Component Fault Trees into the UML. In: Juergen Dingel and Arnor Solberg, Models in Software Engineering. Lecture Notes in Computer Science. Pages 312-327. Springer Berlin / Heidelberg, 2011.

[Ad12] Rasmus Adler, Sören Kemmann, Peter Liggesmeyer, Pascal Schwinn. Model-based Development of a Safety Concept. To be published in: Proceedings of the PSAM11 & ESREL 2012 Conference.

[Ch93] Chudleigh M.: Hazard analysis using HAZOP: A case study. 12th International Conference on Computer Safety, Reliability and Security (SAFECOMP 93), Springer Verlag, pages 99 – 108, 1993.

[De09] Denger, C. SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts. PhD-Thesis, Technical University of Kaiserslautern, 2009

[Do09] D. Domis, M. Forster, S. Kemmann, and M. Trapp. Safety concept trees. In Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual, pages 212 {217, jan. 2009.

[DST05] Vincent Debruyne, Françoise Simonot-Lion and Yvon Trinquet. EAST-ADL: An Architecture Description Language. In: Architecture Description Languages, IFIP The International Federation for Information Processing. Chapter 12, Pages 181–195. Springer Boston, 2005.

[DT08] Dominik Domis, Mario Trapp: Integrating Safety Analyses and Component-Based Design. SAFECOMP 2008: 58-71

[Fe94] P Fenelon, et al, Towards Integrated Safety Analysis and Design, In ACM Applied Computing Review, 2(1):21-32, 1994, ACM Press

[GKS07] P.J. Graydon, J.C. Knight, E.A. Strunk., „Assurance Based Development of Critical Systems“, in Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 347 – 357, 2007.

[LW99] Lutz, R.R., Woodhouse R. M.: Bi-directional Analysis for Certification of Safety-Critical Software. In the proceedings of the International Software Assurance Certification Conference (ISACC 99), Springer Verlag, pages 1 – 9, 1999.

[Pa04] Papadopoulos Y. et al.: Automating the Failure Mode and Effects Analysis of Safety Critical Systems. In the proceedings of the 8th International Symposium on High Assurance Systems Engineering (HASE 2004), pages 310 – 311, 2004.

[PD99] Y. Papadopoulos, J. McDermid, “Hierarchically Performed Hazard Origin and Propagation Studies”, in Proceedings of the 18th International Conference on Computer Safety, Reliability and Security, LNCS 1608:139-152, 1999, Springer Verlag.

[Pu99] Pumfrey D.J.: The Principled Design of Computer System Safety Analysis. PhD thesis. Department of Computer Science, University of York, York, UK, 1999.

[RCC99] Redmill F., Chudleigh M., Catmur J.: System Safety: HAZOP and Software HAZOP. Chichester: John Wiley & Sons Ltd. 248, 1999

[Sc12] Karsten Schmidt, Markus Buhlmann, Christoph Ficek, Kai Richter. Entwurfsaspekte für hochintegrierte Steuergeräte mit unterschiedlichen ASIL –Stufen, ATZ elektronik, 01/2012, Seiten 34-40

[We08] Tim Weilkiens. Systems Engineering mit SysML/UML: Modellierung, Analyse, Design. Dpunkt-Verl., 2., aktualisierte u. erweiterte Edition, Oktober 2008.

[Wi96] S.P. Wilson, J.A. Mcdermid, C.H. Pygott, and D.J. Tombs, "Assessing Complex Computer Based Systems using the Goal Structuring Notation", in ICECCS '96: Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), Washington, DC, USA: IEEE Computer Society, 1996, p. 498.

[1] ISO 26262: Road vehicles - Functional safety.

[2] http://www.audi.de/eperf/brand/de.html

[3] http://spes2020.informatik.tu-muenchen.de/ (Vorgänger von SPES_XT)

[4] http://www.safe-project.eu/

A Systematic Approach to a Quantified Security Risk Analysis for Vehicular IT Systems

Marko Wolf, ESCRYPT GmbH, Munich, Germany (marko.wolf@escrypt.com)
Michael Scheibel, TÜV Informationstechnik GmbH, Essen, Germany (m.scheibel@tuvit.de)

Abstract: By now, security engineering is an accepted challenge in the development of most vehicular IT systems. However, even though many vehicular security threats and effective protection measures are known in general, automotive engineers have difficulties to realize efficient security solutions such that the costs for certain protection measures are appropriate to the actual security threats in order to avoid "under- protection" as well as "over-protection", which both are unacceptable particularly in the automotive domain.

By applying a thorough security risk analysis, which incorporates the special characteristics of the automotive domain, we would have a qualified taxonomy to make well-founded decisions about the security measures effectively required. We therefor present a methodical approach for conducting a meaningful security risk analysis, which focusses particularly on vehicular IT systems. This approach applies systematic estimations for the two mandatory factors of any risk analysis, the potential damages and the probability of a successful security attack, both based on industry-proven methods and taxonomies carefully adapted to vehicular IT security scenarios.

1 Motivation

Strong IT security measures are often mandatory to enforce vehicular business models, liability, legal issues, warranty issues, and in particular to ensure the dependability of many of the next generation vehicular safety systems [An03, Br04, Wo09]. The automotive security protection measures, which will become actually implemented, should be determined by a well-founded costs benefit analysis, which prevents undersized, but also oversized security solutions. Consequently, we need a reliable taxonomy to be able to balance the costs and the complexity added for realizing a vehicular security measure against the potential damage caused by a potential security breach of the particular vehicular IT system. By applying a well-founded security risk analysis, which systematically evaluates the "difficulty" for realizing a certain security attack and the damage caused by this attack, we would have a qualified taxonomy to make well-founded decisions about the security protection measures effectively required. A meaningful security risk analysis thus particularly enables the realization of so-called *economic security* solutions that means that the total cost of a successful attack shall exceed the potential economic gain and shall be proportional to the potential damage, respectively. Finally yet importantly, a meaningful risk analysis would act as the underlying "security business model" to argue and prove for the necessity and necessary strength of appropriate IT security measures.

1.1 Our Contribution

Even though some general approaches and simple taxonomies for IT risk analyses already exist (e.g., [TVRA]), there is so far no systematic approach for IT security risk assessment adapted to vehicular IT scenarios, which are different to standard IT systems, for instance, regarding the attack paths (e.g., internal physical attacker) or regarding potential attack impacts (e.g., driving safety). Hence, this work provides a carefully adapted, four-step methodical approach for calculating the IT security risks especially for vehicular IT scenarios. Our approach therefor provides a systematic evaluation scheme together with an appropriate taxonomy, which enables a quantified rating for the two underlying factors, the "difficulty" for realizing a certain security attack, and the damage that can be caused by this attack, both well-adapted to the special characteristics of the automotive domain. The practical feasibility of our proposed approach will be demonstrated and illustrated by a real-world automotive security application example.

The paper is organized as follows: First, we give some background information on the methods we use for developing our risk analysis approach (Section 2). After that, we describe how these methods can be systematically combined to form a risk analysis method and give an example of its application (Section 3). We then discuss how to get relevant input data for risk analyses and how to use it in our risk analysis model (Section 4). We finally close our contribution with a short note about a helpful tool implementing our approach and assisting the systematic acquisition of all relevant input data (Section 4.3) together with a short summary of our results by giving an outlook and an overview of open questions (Section 5).

1.2 Related Work

Various authors have identified the need for information security and software protection in vehicles [An03, Br04, Wo09]. They present different security vulnerabilities and threats and propose a range of security measures for existing and upcoming vehicular IT systems. However, except for some completely (legally) mandatory security requirements, there are no systematic cost-benefit analyses for efficiently thwarting other security threats. On the other hand, there are already first systematic security risk evaluation approaches for standard IT systems such as telecommunication networks or industry automation [Fi05, TVRA, VDI2182]. However, these approaches are only limitedly applicable to the special security environment of a vehicular IT system due to the completely different attacker incentives, attacker capabilities (e.g., internal physical attacks) and different potential damages (e.g., human injuries, scalability). Moreover, both approaches mentioned remain quite inexplicit and generic especially regarding the identification and estimation of potential damages. After all, [GL02, Wi06] analyze the optimal investment level in information security as the relation of investments to protected assets. As opposed to this universal view to economic security our approach provides a very concrete, quantified view clearly adapted to the special characteristics of the automotive domain.

2 Preliminaries

The following sections give some background information on the risk definition within an IT security context, the Common Criteria for IT Security Evaluation (CC) and the

Safety Integrity Levels (SIL) used in our approach. These are methods from different IT disciplines, not only from the automotive domain. However, they later become specially adapted and linked together to enable a well-founded vehicular security risk evaluation.

2.1 Risk Definition

In most engineering disciplines (e.g., [EN50126]), risk is generally defined as follows.

risk = (probability of an accident) x (expected losses through that accident) (2.1)

For IT security scenarios, the *probability of an accident* can be mapped to the *attack potential* (cf. Section 3.2) required to successfully mount a certain attack, which means to carry out an existing security threat and to misuse a security vulnerability. In our case, the attack potential describes (amongst others) the accumulated technical, financial, and intellectual resources that are required to successfully mount a certain attack. This approach is based on the meaningful assumption that the probability of an accident – which in our IT security scenario means a successful security attack – is decreasing with the increase of the attack potential required. The *expected losses* in turn can be mapped to the *damage potential* (cf. Section 3.3) that means the global sum of all financial and operational damages and particularly of all potential damages regarding vehicular driving safety (if applicable).

2.2 Common Criteria

Most security evaluations of IT (security) products today are done based on the *Common Criteria for Information Technology Security Evaluation* (CC) [CC07]. These criteria basically consist of a catalogue of pre-defined security functional requirements that the evaluation target claims to fulfill as well as pre-defined levels of assurance that give insight on how deep the evaluation of this claim has been done. Custom functional requirements and assurance packages can be added. The evaluation is carried out by an evaluation laboratory and supervised by a governmental institution. As such, evaluations up to a certain evaluation assurance level (EAL) are internationally accepted [CCRA]. The criteria are accompanied by the *Common Methodology for Information Technology Security Evaluation* [CEM] describing the methodology behind a CC security evaluation. The CEM includes a method for calculation the potential of attacks on the evaluation target, which we use for our risk analysis taxonomy.

2.3 Safety Integrity Levels (SIL)

The *Safety Integrity Levels* (SILs) represent a discrete, systematic classification for the functional reliability of safety relevant electronics. There are four SILs, with SIL 4 having the lowest risk for a malfunction (and thus the highest reliability) and SIL 1 in comparison having a “higher” risk for a malfunction (and thus a “lower” reliability). The SILs are defined in [IEC61508] and can be directly referenced to the abbreviated injury scale (MAIS) from the Association for the Advancement of Automotive Medicine (AAAM) [AAA05] that classifies the severity of injuries from car accidents (cf. Table 1). There have been very recent standardization efforts to translate the more generic SIL definitions into specialized *Automotive Safety Integrity Levels* (ASIL) in order to have a more precise adaption of SIL for passenger car vehicular electronics. The corresponding

standard [ISO26262] eases and harmonizes the determination of automotive-specific risk classifications for potential safety hazards caused by malfunctioning vehicular electronics. The metrics and proportions provided by SIL (cf. Table 1) will be directly adapted by our safety damage potential taxonomy, which is described in Section 3.3.

Maximum Abbreviated Injury Scale		SIL reference [IEC61508]	Risk reduction factor (SIL)	ASIL reference [ISO26262]
S0	No injuries	1	10 – 100	A
S1	Light and moderate injuries	2	100 – 1000	B/C
S2	Severe and life-threatening injuries (survival probable)	3	1000 – 10,000	C/D
S3	Life-threatening injuries (survival uncertain), fatal injuries	4	10,000 – 100,000	– (not applicable)

Table 1: Safety integrity level metrics and corresponding proportions

3 Vehicular IT Security Risk Analysis

This section describes our approach to carry out a vehicular IT security risk analysis that is required within virtually all (e.g., [TVRA, VDI2182]) security design cycles (cf. Figure 1). A vehicular IT security risk analysis hence starts with an identification of the actual security assets and their high-level security objectives, which are exposed to certain security threats from potential attack paths (cf. Section 3.1). We then motivate and describe our approaches for calculating the attack potential (AP) and the vehicular damage potential (DP) for a certain attack path (cf. Sections 3.2 and 3.3). Based on AP and DP, we finally motivate and describe our approach to assess and classify the respective vehicular IT security risk. The section closes with a practical example using our approach for a vehicular IT security risk analysis.

3.1 Identification of Security Objectives, Security Threats, and Attack Paths

Before we can evaluate any vehicular attacks, we have to identify the high-level *security objectives* sometimes also referenced as *security goals* or *security aims*. Security objectives accumulate all relevant security assets (e.g., critical data, functionality, or resources) and *security policies* (e.g., "Only authorized personnel may change function parameters of this vehicular component.") together with potential misuse cases and issues to be resolved on a very high-level basis. A security objective for a certain vehicular electronic control unit (ECU) could be, for instance, preventing IP theft or counterfeiting of ECU's software by preserving the confidentiality of the corresponding ECU software. The next step in our approach is to identify possible *security threats* against each of the security objectives we have identified before. A threat is hereby defined by an attacker, an adverse action, and the attacked asset [CC07, Part 1, A.6.2]. A community-developed set of relevant security objectives, security threats, and generic security evaluation criteria for a given family of IT products or IT systems is called a *protection profile* (PP). A PP typically covers all important, known security issues (independently from potential security solutions) that at least have to be verified for

proper counteracting in a security evaluation for evaluation targets of this family. Hence, PPs are often created in context of CC evaluations and are currently available for many IT products and IT systems, but unfortunately not for automotive IT components such as ECUs (Electronic Control Units). By then, we refer to general approaches for identifying security objectives, threats and attack paths (e.g., [Fi05, TVRA, VDI2182]) in combination with corresponding vehicular IT security work (e.g., [An03, Br04, Wo09]).

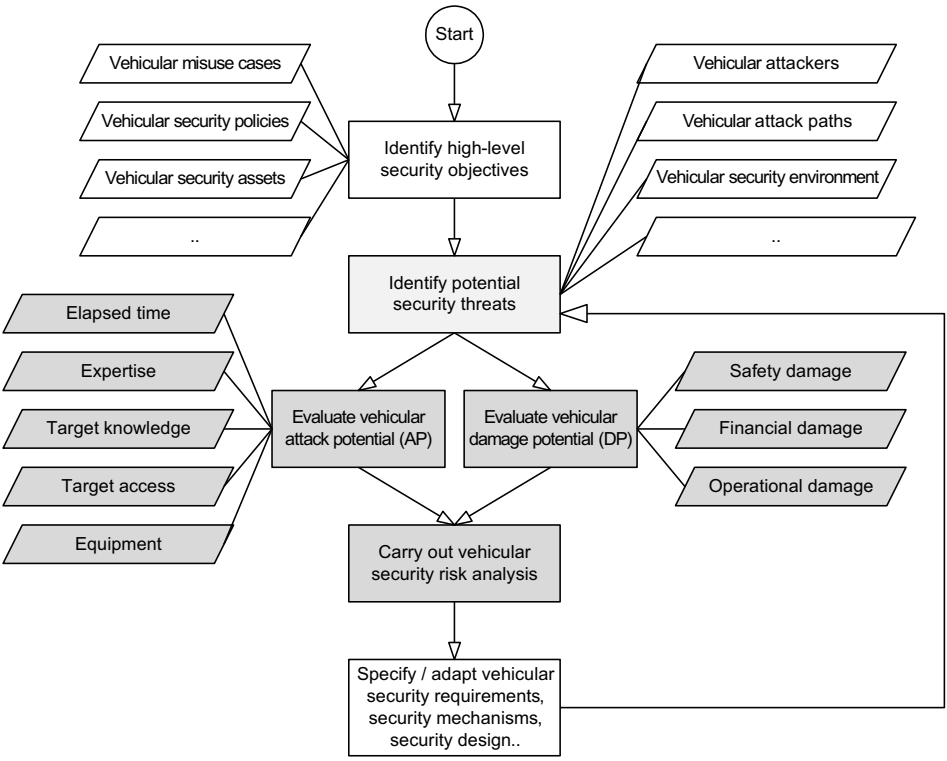


Figure 1: Security risk analysis (gray) as part of the vehicular security design process

The security threats on a security objective can then be organized in an attack tree as shown in Figure 2. Please note that any attack tree is by no means (and cannot be) complete. In contrast to the original attack tree proposal that uses the attacker's goal as the root node [Sc99], we put the security objective at the top. A successful attack on the security objective can then be described by a path from a leaf to the root. Please see [ABD06] and [MSR04] for other applications of attack trees and similar techniques in automotive contexts. We can now calculate the attack potential required to successfully carry out each of the attack paths on our security objective.

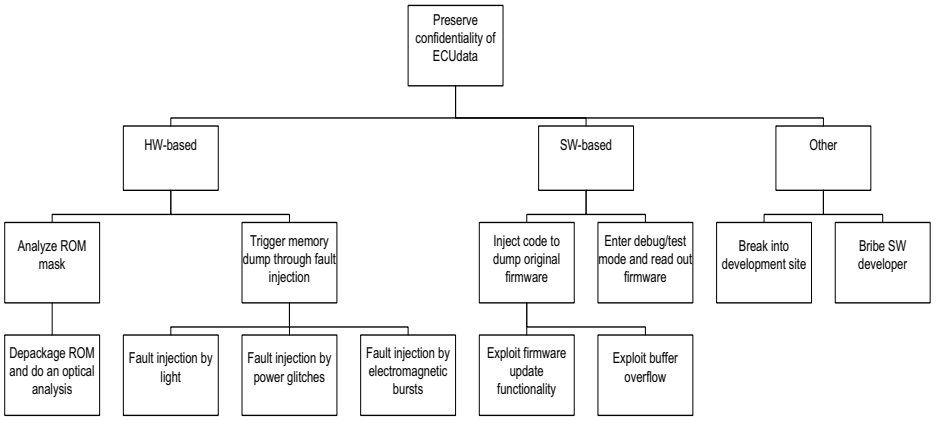


Figure 2: Incomplete attack tree against an ECU security objective regarding data confidentiality

3.2 Attack Potential Calculation

To calculate the attack potential for an identified attack path we follow the industry-proven, standardized CC approach as described in [CEM, B.4] and hence estimate the resources needed to successfully mount the attack in terms of time required for identification and exploitation[1], specialist expertise, knowledge of target, and access and equipment required. This approach can include both known and assumed to be (at least medium-term) possible security attacks (cf. for instance RSA factoring challenges) whereas the latter yields to a higher attack potential. The individual factors are shown in Table 2, which has been taken from [CEM, B.4.2.3] with only minor modifications[2]. All estimations have to consider the worst-case scenario from the security perspective. In case, the attack reference estimation is in-between two categorizations, it is also possible to take an appropriate intermediate value for the corresponding factor.

Category	[CEM, B.4.2.3] Reference	More detailed automotive domain related reference (if applicable)	Factor
Elapsed time	Hours	(none)	0
	Days	(none)	1
	Weeks	(none)	3
	Months	(none)	7
Specialist expertise	Layman	Ordinary vehicle owner/driver; Knows only very simple attacks (e.g., Internet feature activation code generators; simple dip switches or simple shortcuts)	0
	Proficient person	Experienced owner, ordinary garage personnel; Knows simple, popular attacks (e.g., odometer tuning, installing counterfeit parts)	3

[1] Note that CC from version 3.1 no longer distinguishes between identification and exploitation phase.
[2] For example, the category "Elapsed Time" has been carefully reduced from ten to four references.

Category	[CEM, B.4.2.3] Reference	More detailed automotive domain related reference (if applicable)	Factor
	Expert	Specially experienced garage personnel (e.g., <20%); Knows also some more sophisticated, but established attacks (e.g., installing pirate smartcards)	6
	Multiple expert	Highly experienced (garage) personnel (e.g., <1%); Knows also very recent state-of-the-art (academic) attacks (e.g., side-channels, cryptanalysis, zero-day exploit)	8
Knowledge of the target	Public information	Everything that can be found in book stores or in the Internet; Information shared without non-disclosure agreements; "secret" shortcuts (e.g., hot key)	0
	Restricted information	Information shared between different organizations (e.g., OEM and supplier) only under non-disclosure agreements (e.g., source code, internal documentation)	3
	Sensitive information	Information shared only under non-disclosure agreements only within an organization, i.e., exclusive to OEM or to a supplier (e.g., key parameters)	7
	Critical information	Information (traceable) exclusively accessible to only a few persons within an organization (e.g., secret root signing key)	11
Access	Unnecessary or unlimited	Logical or remote access without physical presence, for instance, wireless or via Internet (e.g., V2X or cellular interface, critical vehicle IT backend vulnerability)	0
	Easy	Physical access to interior or exterior but without using any special tools (e.g., opening the hood to access wires, simple removing some car interior lining)	1
	Moderate	Complex disassembly of vehicle parts to access deep internals (e.g., ECU flash memory access) but without breaking sophisticated tamper-protection boundaries (e.g., more than special screws and similar "unsophisticated" measures)	4
	Difficult	Disassembly on microelectronic level (e.g., micro probing/cutting, chemistry) including breaking some sophisticated tamper-protection boundaries	10
Equipment	Standard	Readily available, e.g., common IT device such as notebooks up to simple OBD diagnosis devices; everything a common amateur mechanic could have at home	0
	Specialized	Professional garage equipment, but still (somehow) freely available, e.g., in-vehicle communication devices (e.g., CAN cards) up to costly garage diagnosis equipment	4
	Bespoke	At least one equipment item not freely available such	7

Category	[CEM, B.4.2.3] Reference	More detailed automotive domain related reference (if applicable)	Factor
		as manufacturer-restricted special equipment or equipment with costs > 50,000€ (e.g., electron microscope)	
	Multiple bespoke	More than one bespoke equipment item	9

Table 2: Reference classification for the attack potential factors

The overall attack potential AP is then calculated by estimating and adding the factor for each category as shown in equation (3.1).

$$AP = AP_{time} + AP_{expertise} + AP_{knowledge} + AP_{access} + AP_{equipment} \quad (3.1)$$

Thus, AP represents the accumulated attack potential that merges all resources required for successfully mounting a certain attack path. The total AP value can now be translated into an attacker classification as shown in Table 3. The classification puts AP into the categories as defined by the Common Criteria [CEM, B.4].

AP	Total attack potential classification
0 – 9	Basic
10 – 13	Enhanced Basic
14 – 19	Moderate
20 – 24	High
> 24	Beyond high

Table 3: Attack potential (AP) classification

3.3 Damage Potential Calculation

Our calculation of the damage potential is based on three damage types a successful attack against a certain vehicular security objective can yield to, namely safety damage, financial damage, and/or operational damage (cf. Table 4).

Safety damages include any unfavorable incident that might cause injuries to vehicle passengers as result of a successful security attack. For the safety damage classification, we use the industry-proven ASIL classification from [ISO26262] (cf. Section 2.3). For the quantitative classification of the corresponding factors, we also transferred the (A)SIL decimal power scaling. However, note that the proposed safety damage factors solely represent arithmetical values without any ethical rating (e.g., in comparison with the financial damage factors).

Financial damages include the global sum of all losses as result of a successful security attack that do not directly reflect a safety issue. Therefore, financial damages include all *direct financial losses*, for instance, all financial losses from broken business models (e.g., broken after-sales feature activation), legal implications (e.g., penalties), product liability issues (e.g., penalties or callbacks), counterfeiting, but also estimations for all *indirect financial losses*, for instance, regarding business reputation damages or loss of

market shares. However, since the meaning of some concrete sums of money, can be very different for every company, the given financial damage classifications are related to the general financial damage classifications from the "IT-Grundschutz" [BSI-100-4] issued by the German Information Security Agency. To account the – in contrast to most safety damages – again comparatively smaller consequences properly, we shifted the corresponding factors in proportion to the safety damage classes downwards by one magnitude (while keeping the decimal power scaling).

Operational damages in turn, include all other unfavorable incidents, which do not directly cause any injuries and do not have any meaningful measurable financial dimension. This includes, for instance, damages to the vehicle functionality (e.g., breakdown of the air-condition system) up to potential damages to vehicular infrastructures (e.g., traffic management systems). The classification is again based on a shortened, but industry-proven estimation as used in vehicular defect severity categorization already such as FMEA (Failure Mode and Effects Analysis) [AIA08]. To account the – in contrast to the safety damages and the chosen financial categories – comparatively smaller consequences, we shifted the corresponding factors again downwards by one magnitude (while keeping the decimal power scaling).

Damage category	**Damage reference**	**Factor**
Safety severity classes	Life-threatening injuries (survival uncertain), fatal injuries	10,000
	Severe and life-threatening injuries (survival probable)	1,000
	Light and moderate injuries	100
	No injuries	0
Finance severity classes (global sum)	Existence-threatening financial damage (e.g., monetary damage is >30% of annual sales)	1,000
	Substantial financial damage, but yet not existence-threatening (e.g., monetary damage is 20% – 30% of annual sales)	100
	Undesirable financial damage (e.g., monetary damage is 5% – 20% of annual sales)	10
	No or tolerable financial damage (e.g., monetary damage is <5% of annual sales)	0
Operational functionality severity classes	Vehicles unusable, i.e., one or more fundamental functions are affected. The vehicle usage is infeasible. This can be compared with FMEA severity rating above 8.	100
	Service required, i.e., an important function is affected. The vehicle can be used only with massive restrictions. This can be compared with FMEA severity rating 6 to 8.	10
	Comfort affected, i.e., a function is affected. The vehicle can be used with some restrictions. This can be compared with FMEA severity rating 2 to 5.	1
	No relevant effects, i.e., at most, an unimportant function is affected and the vehicle can be used without restrictions. This can be compared with FMEA severity rating 1.	0

Table 4: Reference classification for the damage potential factors

The total damage potential DP can then be calculated by estimating and adding the values of the three individual factors (cf. Section 4 on estimation methodology) as shown in equation (3.2). Similar to the attack potential calculations, all estimations have to consider the worst-case scenario. In case, the damage reference estimation is in-between two categorizations, it is also possible to take an appropriate intermediate value for the corresponding damage factor.

$$DP = DP_{safety} + DP_{financial} + DP_{operational} \quad (3.2)$$

Thus, DP represents the accumulated damage potential that merges all potential financial and non-financial consequences caused by a successful security attack against a certain security objective, regardless of the occurrence likelihood / difficulty for such an attack. The underlying decimal power weighting of the evaluation intervals of the three damage categories (i.e., safety, financial, and operational) is shown in Figure 3.

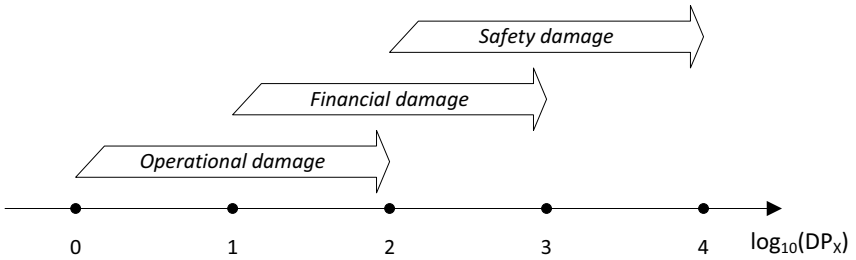


Figure 3: Damage interval relations

Note that a successfully violated security objective through a successfully accomplished security attack often may yield to different autonomous consequences regarding safety, finance, or vehicle operation in parallel. However to ease application, we suggest to consider from all potential consequences only the worst consequence for each category, which includes all other "lower damages". For example, a successful security attack, which violates a security objective that "only authorized personnel may modify the ECU firmware" and hence enables unauthorized modifications, may have three individual, autonomous consequences. It could yield to two independent safety damages (e.g., due to two potential ECU malfunctions) but it could also yield to a certain financial damage independent from the two safety damages (e.g., due to an unauthorized feature activation). For calculating the corresponding damage factor, we would first look for the consequence with the worst-case safety damage and would take only this consequence for determining DP_{safety}. We would then look for the consequence with the worst-case financial damage and would take this consequence for determining $DP_{financial}$. Similarly, we would look for the worst-case operational damage consequence and would take this consequence for determining $DP_{operational}$. The sum of all worst-case factors would then determine the total DP caused by the corresponding security objective violation.

Finally, the total DP value can be translated into damage categories as shown in Table 5. These classifications use the well-established decimal power scaling (cf. Table 1),

realized in such a way that severe injuries and very high financial damages result in a classification of catastrophic.

DP	Total damage potential classification
0 – 2	Insignificant
3 – 21	Medium
22 – 210	Critical
> 210	Catastrophic

Table 5: Damage potential (DP) classification

3.4 Risk Assessment

To conclude our risk analysis, we finally create a so-called *risk matrix*, which combines the probability of a successful security attack path (attack potential AP) and the expected losses of that incident (damage potential DP) to a meaningful risk classification.

Unfortunately, both, [IEC61508] as well as its automotive derivation [ISO26262] (currently) do not provide any obligatory risk assessment method or risk acceptance values. This has mainly political reasons, since every industry (or even each manufacturer) that applies the generic [IEC61508] standard would have its own individual definitions of acceptable and non-acceptable risks. However, we found that the risk matrix and the corresponding risks assessments according to [EN50126] – originating from the railway safety engineering – satisfactorily fits our needs. Hence, Table 6 serves as at least as a basis for an individually adaptable automotive IT security risk acceptance matrix. A further, however general security risk taxonomy from the research perspective can be found in [Fi05].

AP↓	Probability reference	Risk assessment			
Basic	Certain	Undesirable	Inacceptable	Inacceptable	Inacceptable
Enhanced Basic	Likely	Tolerable	Undesirable	Inacceptable	Inacceptable
Moderate	Possibly	Tolerable	Undesirable	Inacceptable	Inacceptable
High	Unlikely	Negligible	Tolerable	Undesirable	Inacceptable
Beyond High	Rare	Negligible	Negligible	Tolerable	Inacceptable
	Practically infeasible	Negligible	Negligible	Negligible	Undesirable
DP →		Insignificant	Medium	Critical	Catastrophic

Table 6: Exemplary automotive IT security risk matrix

3.5 Application Example

We now illustrate our approach by giving a brief example. We consider a customer modifying a safety-critical ECU firmware for an unauthorized performance manipulation of his car's engine (i.e., "*chip tuning*"). This attack aims at violating the security objective that the integrity of the ECU firmware shall be protected. In our example,

uploading custom code to the ECU is a protected capability, which means, only authentic original and – for IP protection reasons – encrypted code will be accepted by the ECU. We assume that the protection mechanism is not enforced by a security chip, but solely through software security measures. Thus, the attacker is able to reverse-engineer the firmware and/or gets to know non-public information in order to do the custom code software update. This attack path could result in the following attack potential rating according to method described in Section 3.2.

The time elapsed to collect necessary knowledge, to obtain necessary equipment, and to actually execute the attack probably will be between *days* and *weeks* (Factor 2). Note that it is perfectly alright two choose intermediate factors for an estimation of an AP category, such as factor 2 for a rating which is between the two references "days" and "weeks". The expertise required to identify and carry out such an attack for the first time would be *expert* (Factor 6). The necessary knowledge of the target will be rated *restricted* (Factor 3), since at least basic knowledge of the internal software structure and functionality will be required to succeed between days and weeks. As the attacker usually is also the legitimate owner of the vehicle, he has full access to his car all the time such that the window of opportunity is practically unlimited. However, the attacker has to communicate somehow with the ECU, which requires at least *easy* physical access (Factor 1). Finally, standard equipment plus some additional software/cables may be required to carry out the attack. However, those specialized parts can be easily obtained via the Internet so that the necessary equipment is rated to between *standard* and *specialized* (Factor 3). The calculation of the total AP is summarized in Table 7 and results in a required AP of *moderate*. However, once an attack can be automated and the required code (a so-called "exploit") is published on the Internet, its rating usually decreases significantly and results in a new attack path.

We now analyze the damage potential if the corresponding security objective has been violated by this attack path. If some safety-critical code has been modified in an unauthorized way, its execution may lead – from the worst-case safety perspective – to severe accidents even with injuries (Factor 100). From the worst-case financial perspective, which are not safety-related, (undetected) custom software installations may lead to spurious warranty claims or false liability issues yielding to a financial damage is rated to be at most 5% to 20% of the related annual sales (Factor 10). Finally, an unauthorized modified software code may also – independently from any safety or financial consequences – lead to some impacts or failures of certain vehicular functionality, since it has not been verified by the manufacturer (Factor 5). As can be seen in Table 7 the overall damage potential is calculated and categorized as *critical*.

For the final IT security risk assessment of this attack path, we apply our vehicular IT security risk matrix as defined in Table 6 that rates this attack path as an *undesirable* security risk (i.e., 1/*moderate* x *critical*). Implementing better security measures into the ECU such as a hardware-based integrity protection mechanism [SSW08] as opposed to a software-only mechanism may significantly increase the attack potential required and thereby decrease the overall risk for instance to a *tolerable* level.

Please note that there may be several reasonable calculations of the attack potential because the prerequisites for the attack may differ. As an example, there usually is a trade-off between expertise and time that means an attacker needs either much expertise

or much time. In these cases, it is safe to choose the lowest total AP factor from the set of attack scenarios. Moreover, some attacks cannot be countered directly by the IT component for which the risk analysis is being done (e.g., security during development). These attacks may be non-technical, such as social engineering, or target technical systems that are out-of-scope for the manufacturer of the IT component. A formal way of incorporating these attacks into the risk analysis is to define assumptions under which the risk analysis holds, for instance, to assume the development site is secure.

AP/DP category	**Reference**	**Factor**
Elapsed time	*Between days and weeks*	2
Specialist expertise	*Expert*	6
Target knowledge	*Restricted information*	3
Access	*Unlimited*	1
Equipment	*Standard/specialized*	3
Total AP	*Moderate*	15
Safety severity	*In worst case up to light and moderate injuries*	100
Finance severity	*Undesirable, i.e., at most 5% – 20% of related annual sales*	10
Operational severity	*Vehicle operable, but increased chance for inherent failures*	5
Total DP	*Critical*	115
Sec. Risk = 1/AP x DP	***1/moderate*(15) x *critical* (115) = *Undesirable***	

Table 7: Exemplary calculation of attack potential, damage potential, and resulting security risk

4 Getting Relevant Data

An important question is how to get all the relevant data in order to (i) identify possible attacks and potentially resulting damages and (ii) to classify damage factors and attack factors accordingly. For the identification of potential security objectives and possible attack paths, we suggest the use of security questionnaires (cf. Section 4.1). For the identification and classification of attacks, we apply the approach as described in Section 4.2. Lastly, for the estimation of potential damages according to Section 3.3, we suggest the use of more detailed security questionnaires that include also analyses of former similar security incidents.

4.1 Security Questionnaire

In the following, we provide a draft proposal for a security questionnaire in order to collect the information necessary to conclude a meaningful security risk analysis. This includes a basic understanding of application usage (1), the identification of potential security objectives (2), collection of potential attacks and misuse cases known (3), and identification of already existing security measures (4). This approach is similar to the Common Criteria model of defining a "Security Target" describing the evaluation target, its security environment, its security objectives, and its functional security requirements.

(1) Please give a short description of your vehicular IT application including all involved entities (e.g., driver, OEM, service, third parties) and all (security) relevant use cases!

Main components, available interfaces, general functionalities, entities, security environment, and corresponding use cases
(2) Please provide data, resources, or functionality where certain entities may have certain security objectives!
Entity requirements on data regarding confidentiality, integrity, authenticity, availability, freshness, or access control. Entity requirements on functionality and resources regarding correctness, availability, access control, or quality-of-service control
(3) Please describe shortly any attacks, misuse cases and attack intentions you know for this vehicular IT application!
Attack goal, misuse case, attack path, attacker model, attack tree
(4) Please describe shortly any security measures that possibly already exist to enforce a security objective from (2)!
(a) Security objective 1: Security protection measure 1, Security protection measure 2 (b) Security objective 2: Security protection measure 1 (c) …
(5) Please provide estimations about the preliminaries and costs for each attack identified in (4)!
Reference to: Table 3: Attack potential (AP) classification
(6) Please provide estimations about the potential damages for each attack identified in (4)!
Reference to: Table 5: Damage potential (DP) classification

Table 8: Exemplary IT security questionnaire for risk analyses

For a fine-grained estimation of attack and damage potentials, the security questionnaire should use the categories provided by our estimation method as proposed in Section 3.2 and Section 3.3, respectively. In case of a new security-relevant application, the damage categories can be estimated only on approximations of the corresponding experts. In case the security-relevant application is already target of security attacks, the estimations would be much more precise as they can include the severity and frequency of already existing incidents. Thus, we recommend that the responsible developer completes this questionnaire together with a security expert and with experts capable to estimate the safety, financial, and operational consequences caused by a security breach of the analyzed vehicular application.

4.2 Identification and classification of attacks

The Common Criteria require the evaluator to examine public sources as well as evaluation evidence (i.e. internal documentation) to identify potential vulnerabilities and to assess the possibility of their exploitation. If a potential vulnerability is exploitable in the intended operational environment of the evaluation target, and the attack potential required for a successful identification and exploitation is beyond the level the evaluation target claims to resist, a security solution has to be implemented. The assessment of the attack factors are done in accordance with the CC and the knowledge and experience of the evaluator. Thus, once an IT product has been CC-certified, reliable statements about the resistant to attacks on its security objectives can be made. The identification and classification of possible attacks, which has to be done in context of our risk analysis methodology, can be based on CC certification results. Furthermore, a CC certification of a security-critical vehicular system gives internationally accepted

assurance that the manufacturer has put serious efforts on averting damages resulting from an attack on the system.

4.3 Implementation

We have implemented our approach as a simple spreadsheet application that allows a systematic acquisition of security targets, their possible attack paths, and their individual damage consequences. It enables estimations of attack and damages potentials according to our proposed approach, and provides an automatic risk assessment according to our adapted [EN50126] security risk matrix.

In doing so, our approach has already been successfully applied in several security analysis and consulting projects for several well-known automotive OEMs.

5 Summary and Outlook

In this paper, we presented a methodical approach to balance the security costs for implementing vehicular security measures against the security risks of corresponding automotive security attacks. This approach is based on well-established methodologies, which have been carefully adapted and combined for efficient, meaningful application in vehicular IT security scenarios. By integrating the special characteristics of the automotive domain, the proposed approach, provides automotive developers and manufacturers with a quantified assessment methodology for vehicular IT security measures where the trade-off between costs and security risks can be clearly analyzed to enable well-founded decisions. Our approach is based on the assumption that the probability of a successful attack on a security measure is decreasing with the increase of the attack potential required (see Table 6). This is only true for most but not all real-world scenarios. As an example, imagine a car manufacturer that – no matter the cost – is willing to reverse-engineer the software of a brand-new competitors' product. How to get rid of this assumption by extending our taxonomy will be subject of further research. Having more fine-grained factors or a better weighting of the individual factors and categories, would be another subject of further research. Especially, first practical results from our proposed security questionnaires could provide valuable input for verifying or adjusting our scales to current automotive industry real world scenarios. As mentioned in Section 3.1, an automotive-specific Protection Profile (PP) covering all important, known ECU security issues (independently from potential security solutions) evaluation, would clearly ease and systematize the identification of vehicular security objectives, threats, and attack paths. Another interesting question is how our results relate to the discussions of [GL02] and [Wi06] who try to determine the optimal level of investments in information security. According to their work and given requirements, 36.8% to 50% of total cost of protected assets should be invested in security measures. Does our approach yield similar results? In our special context of vehicle mass production, the total investment costs are significantly influenced by the cost per production unit. To (amongst others) get answers on this question, our approach for vehicular IT risk analysis is currently (independently) applied and evaluated for two individual security solutions of two German car manufacturers. Finally, we would like to stress that – even if based on well-found analyses – a risk analysis remains a statistical estimation that inherently includes uncertainties.

References

[AAA05] American Association for Automotive Medicine (AAAM), „The Abbreviated Injury Scale (AIS)", 2005.

[ABD06] Amer Aijaz, Bernd Bochow, Florian Dötzer, Andreas Festag, Matthias Gerlach, Rainer Kroh and Tim Leinmüller, "Attacks on Inter-Vehicle Communication Systems – An Analysis", In *3rd International Workshop on Intelligent Transportation (WIT 2006)*, March 2006.

[AIA08] Automotive Industry Action Group (AIAG), "Potential Failure Mode and Effects Analysis (FMEA)", 2008.

[An03] Ross Anderson, "Electronic Safety and Security – New Challenges for the Car Industry. In *1st Workshop on Embedded Security in Cars (escar)*, Bochum, Germany, November 2003.

[Br04] Manfred Broy, "Sichere Software im Automobil – Potenziale, Herausforderungen, Trends", In *2nd Workshop on Embedded Security in Cars (escar)*, Bochum, Germany, November 2004.

[BSI-100-4] German Information Security Agency. BSI-Standard 100-4: Business Continuity Management, 2008.

[CC07] Common Criteria for Information Technology Security Evaluation, continually maintained as ISO 15408.
"Part 1: Introduction and general model", V.3.1, Revision 1, September 2006,
"Part 2: Security functional components", V.3.1, Revision 2, September 2007,
"Part 3: Security assurance components", V.3.1, Revision 2, September 2007.

[CCRA] Common Criteria Recognition Agreement, www.commoncriteriaportal.org/theccra.html, 2006.

[CEM] Common Methodology for Information Technology Security Evaluation, V.3.1, Revision 2, September 2007, Continually maintained as ISO 18045.

[EN50126] EN 50126, "Railway applications – The specification and demonstration of reliability, availability, maintainability, and safety (RAMS)", 1999.

[Fi05] Donald G. Firesmith, "A Taxonomy of Security-Related Requirements", 2005.

[GL02] Lawrence A. Gordon and Martin P. Loeb, "The economics of information security investment". In *ACM Transactions on Information and System Security*, November 2002.

[IEC61508] IEC 61508, "Functional safety of electrical/electronic/programmable electronic safety-related systems", 2005.

[ISO26262] ISO 26262, "Road vehicles – Functional safety", November 2011.

[MSR04] Stephan Merk, Kathrin Scheidemann, Michael Rudorfer, Thomas Stauner, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, "Security for Downloadable Automotive Services", In In *2nd Workshop on Embedded Security in Cars (escar)*, Bochum, November 2004.

[SSW08] Michael Scheibel, Christian Stüble, Marko Wolf, "An Interoperable Security Architecture for Vehicular Software Protection", In International Workshop on Interoperable Vehicles (IOV 2008), ETH Zürich, Switzerland, March 2008.

[Sc99] Bruce Schneier, "Attack trees – Modelling security threats", In Dr. Dobb's journal, http://www.schneier.com/paper-attacktrees-ddj-ft.html, December 1999.

[TVRA] ETSI TS 102 165-1, "Method and proforma for Threat Vulnerability and Risk Analysis (TVRA)", December 2006.

[VDI2182] VDI/VDE 2182, "IT-security for industrial automation - General model", August 2007.

[Wi06] Jan Willemson, "On the Gordon&Loeb Model for Information Security Investment", In *5th Workshop on the Economics of Information Security (WEIS 2006)*, June 2006.

[Wo09] Marko Wolf, "Security Engineering for Vehicular IT Systems — Improving Trustworthiness and Dependability of Automotive IT Applications", Dissertation, Vieweg+Teubner-Verlag, 2009.

Risk-based Testing of Bluetooth Functionality in an Automotive Environment

Felix Jakob[1], Wolfgang Kremer[1], Andreas Schulze[1,]
Jürgen Großmann[2], Nadja Menz[2], Martin Schneider[2],
Alain-Georges Vouffo Feudjio[2]

[1]Dornier Consulting, Sindelfingen, Germany
[2]Fraunhofer FOKUS, Berlin, Germany

Abstract: This paper describes an approach for risk-based testing of Bluetooth functionality in an automotive environment, recently studied as part of the ITEA-2 research project DIAMONDS.

In the past two decades the functionality of infotainment systems of a modern car increased in complexity. Based on the worldwide evolution towards an information age the driver requests permanently increasing number of functionalities for infotainment systems where the vehicle becomes a connected vehicle which is always on. For instance next generation infotainment systems will provide internet to the cabin and will have integrated of applications like Twitter and Facebook.

These additional online services can be realized by integrating the driver's smartphone with the vehicle infotainment system whereby the phone becomes a logical part of the infotainment head unit. This intense integration of consumer mobile devices with the vehicle electronics and network can also have a negative impact on the security of the vehicle electronics. Even the safety level of the entire automotive system can be affected, which recent studies have shown [Ko10, Ko11].

With respect to these increasing security risks for connected vehicles and future telematics applications, efficient and structured methods are required to verify such systems concerning their robustness against security attacks. One important approach is a risk oriented model-based testing concept as developed in the research project DIAMONDS[1]. Within this project a case study has been defined by Dornier Consulting for the automotive domain focused on connected car applications such as the integration of mobile consumer devices with the in-vehicle network and electronics.

Based on risk analysis a security related prioritization of messages is achieved, which enables a smart and efficient generation of the most relevant test scenarios. Using adapted security test generation methods based on known concepts, such as

[1] ITEA (Information Technology for European Advancement) DIAMONDS project, http://www.itea2-diamonds.org

fuzzing of these pre-defined test scenarios, will then be used to generate the final set of test cases.

This paper discusses the generation process for risk based security test cases using this model-based testing approach, which pays attention to the analysed security flaws. An approach that refers to the generation of test cases that will cover possible unknown flaws by integrating fuzzing methodologies will also be described. The practical implementation of this generation process will be demonstrated by Dornier Consulting's extended Model-Based Testing (MBT) framework do.ATOMS for industrial and commercial application.

1 Introduction

As Information and Communication Technology (ICT) systems become more and more part of our daily lives, current and future vehicles are more and more integrated into ICT networks. The consumer's smartphones, multimedia devices etc. are linked to the vehicles and allow the driver or passengers to use the internet, to access their private phone books or to run their individual applications through the vehicle's integrated infotainment systems.

Today's most common technology to link consumer devices to in-vehicle electronics is Bluetooth, which latest version is 4.0. Such a wireless connection provides the most comfortable way for the driver to access a variety of services. However such a wireless connection implies also the risk of possible misuse which leads to enhanced security issues and risks for the automotive electronics and with that for the passengers.

Koscher et al. [Ko10] showed how today's vehicles can be hacked via security flaws and how these security flaws can even lead to safety critical scenarios for the passengers.

Recently the complexity of ICT systems and automotive electronics increases dramatically and requires modern verification and validation methods, which allow handling of complex systems fast and efficiently. One important approach is the Model-Based Testing (MBT) concept where even complex test scenarios and logics can be defined in models by using the Unified Modelling language (UML), the System Modelling Language (SysML) or other 4th generation languages. State-of-the-art concepts are described in Chapter 2.

Following this MBT approach for complex systems, security aspects of ICT systems and automotive electronics should also be integrated into such MBT methods. In 2010 the European ITEA-2 research project DIAMONDS has been founded by 23 industry and research partners from six countries to develop new methodologies and techniques for model-based security testing and to implement those new concepts into commercial tools for the industry. The DIAMONDS project is very much organized along so called case studies where the research results will be implemented and demonstrated in real industry applications. Preliminary results of the project are stated in Chapter 3.

One case study within the DIAMONDS project has been defined by Dornier Consulting for the automotive domain. It focus on an embedded connectivity module as part of an actual vehicle infotainment system which supports today's smartphones via Bluetooth link for various services/profiles such as hands-free phone, phone book access etc. This automotive case study demonstrates how the new developed model-based security testing concept and tools can be used in the automotive industry.

In Chapter 4 of this paper the first results and findings from the automotive case study in DIAMONDS are shown.

2 State-of-the-Art

2.1 Security in the automotive domain

Security testing in the automotive domain is quite a new discipline. Despite the fact that modern cars provide a wide variety of comfort or multimedia services the corresponding testing measures ensuring the service quality have been mainly taken from a more functional aspect or have been related to passenger safety. Especially the increasing number of wireless communication interfaces (e.g. for internet based services or related to car to car or car to infrastructure communication) lead to the fact that the attack surface of a modern car has increased quite dramatically (as can be seen in the Figure 1).

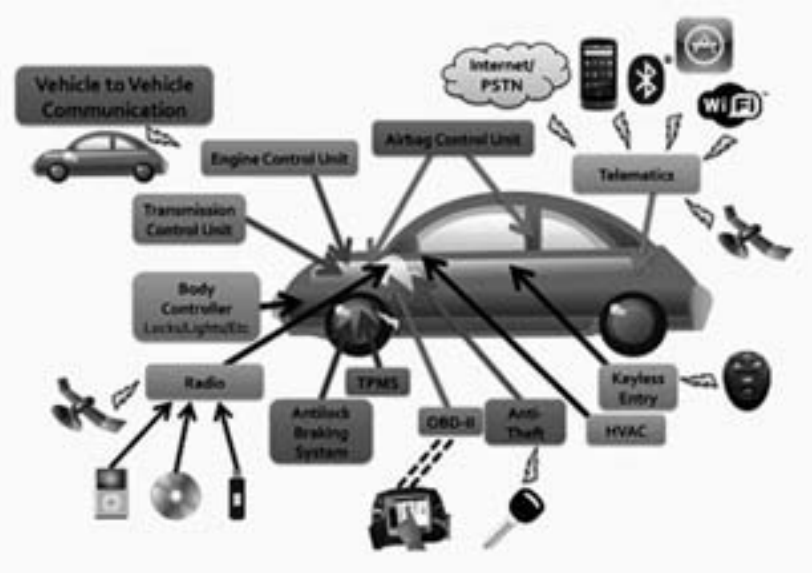


Figure 1: Digital I/O channels appearing on a modern vehicle (from [Ko11])

The main danger that arises from this initial position is the possibility that an attacker, who has no direct (in terms of "physical") access to the car, can nonetheless gain access to the car's interior electronic components. The communication of the car's internal Electronic Control Units (ECUs) relies on various bus systems (like CAN (Controller Area Network), LIN (Local Interconnect Network), or MOST (Media Oriented Systems Transport)) and it should be noted explicitly that the technical design of these bus systems alone does not provide any security features at all. For example, if someone is able to gain direct access to the car's CAN bus communication he can control all car components that are interconnected with the infected component (e.g. by successfully hijacking an ECU like a head unit by using exploits that are accidentally provided by an active Bluetooth connection of the head unit). This threat scenario can be considered as the definitive worst case since an attacker could potentially use the brake control unit to block the left front wheel completely causing the car to spin while driving on the motorway at high speed [Ko10, Ko11].

2.2 Risk-Based Security Testing

Security testing aims for finding any weakness or potential vulnerability that may result from poor or improper system configuration, known and/or unknown hardware or software flaws, or operational weaknesses in process or technical countermeasures. Unfortunately security testing, especially security vulnerability or penetration testing, lacks systematic approaches, which enable the efficient and goal oriented identification, selection, and execution of test cases. Similarly, classical test approaches address risks implicitly rather than systematically. Systems, functions, or modules, which are known to be critical, are tested more intensively than others. The basis of such a test planning is often a very simple and unstructured risk assessment, which usually is performed during or in preparation of the test process. In contrast, there are a number of systematic approaches that can be summarized under the term Risk-Based Testing (RBT) in general and under the term Risk-Based Security Testing (RBST) when security risks are directly addressed.

RBT approaches have in common that the identified systemic risks become the central factor that is used either for a systematic identification of test objectives, for scoring and selecting test cases, for the evaluation of the test quality or for assessing the test results. For example Redmill [Re04, Re05] describes the use of risk analysis to optimize the test planning. The quantified risks are used to prioritize the testing and the analysis of system failures and their impact. Amland [Am00] calculated the risk of the individual software functions and the expected failure probability for these functions weighted with risk factors such as design quality, size, and complexity of the evaluated software. Finally Zimmermann et al. [Zi09] and Stallbaum et al. [SMP08] describe model-based approaches, which allow the automatic derivation of system test cases from different kind of functional models as well as their prioritization based on risk. The latter is called RiteDAP (Risk-based test case Derivation And Prioritization) and uses activity diagrams

to automatically generate test cases. The models are annotated with risk values that are used for the test case prioritization.

Risk-Based Security Testing characterize a methodology that makes software security risks the guiding factor to solve decision problems during testing, e.g. the selection and prioritization of test cases. General technical recommendations on security testing techniques [He03, MTL09] already propose the use of risk analysis results to guide security testing. However, these recommendations are very general in nature and describe in this sense no real methods for risk-based testing. A more detailed approach is described in [Ze11]. The model-based approach for risk-based security testing of cloud environments is based on so called negative security requirements. These requirements are derived from risk analysis in order to cover those security aspects not covered by existing positive requirements. Misuse cases are then defined based on the negative requirements.

One of the key challenges of security testing in general is the detection of potential weaknesses and vulnerabilities on the system under test (SUT) to evaluate the associated risks as accurately as possible. Fuzz testing or fuzzing is a black box testing technique that is broadly used for that purpose. It consists of injecting invalid input data into a system and assessing how it would react [Be11]. The aim is to find deviations of the SUT to its specification (unexpected behaviour, errors, and weaknesses) that lead to vulnerabilities because invalid input is processed instead of being rejected. Such deviations may lead to undefined states of the SUT and can be exploited by an attacker, e.g. by successfully performing a denial-of-service attack because the SUT is crashing or hanging [TDM08]. Because it relies on injecting invalid input data to the SUT, fuzzing is often referred to as negative testing or as interface robustness testing.

Depending on the amount of knowledge about the SUT's protocol, there are different categories of fuzzers, from dumb fuzzers that nearly know nothing about the protocol up to smart fuzzers that use a model of the protocol for communicating with the SUT. The more a fuzzer knows about the protocol, the more efficient is the fuzzer in finding errors [Ta09]. Original fuzzers randomly generate input data for the SUT. Because of this, most generated input data will be rejected because it is invalid in such a large portion that the SUT won't parse it. Hence, most test cases that inject randomly generated data will not find bugs because the invalid data is not processed by the SUT. For that reason, the goal is to generate so called semi-valid input data meaning that the input data is invalid only to a small extend. In order to achieve this goal knowledge of the format of valid input data is needed. A smart fuzzer can employ this knowledge to generate semi-valid input data that may pass the SUT's parser and penetrates deeper into the SUT [Oe05, Ta09]. There, the semi-valid data is processed and may expose security-related weaknesses in the implementation. Additionally, model-based fuzzers can user their knowledge of the protocol behaviour to generate semi-valid input data only for certain messages in certain protocol states, e.g. after authentication.

A big issue while fuzzing is how to determine that a weakness was found. In functional testing, for each input data there exists an expected output that can be compared to the actual output of the SUT. Because fuzzing uses invalid input data to find security-related weaknesses, the expected reaction of the SUT is difficult to determine. At best, it is

specified how the SUT should react on invalid input. But if the SUT does not reject the invalid input data that does not necessarily mean that a weakness was found. Obvious weaknesses or vulnerabilities that can be easily detected are a crash or a hang of the SUT. But a weakness does not need to be exposed immediately after submitting invalid input data. Abnormal behaviour of the SUT can occur much later and it can be more subtle than a crash, e.g. performance degradation or increased memory consumption.

A simple approach in finding hints whether the SUT could be affected by a weakness stimulated by invalid input data is valid case instrumentation [TDM08]. After each fuzz test case a functional test case is executed to determine if the SUT behaves normally. If not, the previous fuzz test case probably found a weakness. Monitoring of the SUT can also assist in finding more subtle weaknesses.

3 Risk-Based Model-Based Testing

The DIAMONDS project develops a risk-based model-based security testing methodology being demonstrated on case studies from a variety of application domains, including the banking and transport domain. Risk-based testing approaches can help to optimize the risk analysis and the risk assessment itself. Risk analysis and risk assessment, similar to other development activities in early project phases, are mainly based on assumptions on the system itself. Risk-based testing results can act as an indicator for the correctness of the assumptions that have been made during the risk analysis and risk assessment. In particular, risk-based testing can help in

- reducing the likelihood of security incidents by discovering unknown vulnerabilities,
- optimizing risk analysis by identifying new risk factors and reassessing the probabilities of risks and their consequences,
- providing further evidence on the functional correctness of countermeasures.

Risk-based model-based security testing is an approach being developed in the DIAMONDS project, combining risk analysis and MBT techniques to assess the security properties of a software-based system. The key goal of the approach is to allow both process groups, i.e. risk analysis and testing to benefit from each other in a consistent and systematic manner. As illustrated by Figure 2, the risk-based model-based security testing process can involve several iterations whereby risk-analysis always plays a central role.

Figure 2: Overview of the Risk-Based Model-Based Security Testing Process

The next sections describe each of the phases of this process.

3.1 Risk Analysis

The DIAMONDS approach proposes to use multiple well-known sources as input for the risk analysis phase: Weakness and vulnerability enumerations from known bodies (e.g. MITRE's CWE, and CVE), attack pattern enumerations as well as security standards (e.g. ISO 27000 series). This will result in a well-defined CORAS threat model including assets, threat scenarios, and vulnerabilities. The CORAS language [LSS11] integrates different tree-based approaches for risk modelling. It is a graph-based modelling approach that emphasizes the modelling of threat scenarios and provides formalisms to annotate the threat scenarios with probability values and formalisms to reason with these annotations.

3.2 Test Model Design

To find an optimal set of test cases, one requires an appropriate identification and selection strategy. Such a strategy on the one hand needs to take into account the available test budget and on the other hand to provide, as far as possible, the necessary test coverage. In functional testing, coverage is often described by the coverage of requirements or the coverage of model elements such as states, transitions or decisions. In risk based testing the aim is for coverage of the identified risks of a system. Yet, security testing often lacks systematic approaches that enable the efficient and goal oriented identification, selection, and execution of test cases.

Within DIAMONDS, a framework was developed to resolve this problem. RBMBT is used to characterize a methodology that makes software risks the guiding factor to solve decision problems during testing, e.g. the selection and prioritization of test cases. A comprehensive risk assessment introduces the notion of risk values, that is the estimation of probabilities and consequences for certain threat scenarios. These risk values are then used to weight threat scenarios and thus help identifying which threat scenarios are more relevant and thus identifying the threat scenarios that are the ones that need to be treated

and tested more carefully. The DIAMONDS framework therefore allows to systematically capture security risks (i.e. vulnerabilities and threat scenarios), treatments and risk values and trace them to test artefacts so that test identification and test selection within the test model design is effectively supported. The implementation of this test case prioritization approach as part of the Dornier Consulting case study is described in Chapter 4.2.

3.3 Test Generation

In addition to traditional fuzzing – with the selection of the messages depending on the message prioritization from the Test Model Design as described in Chapter 3.2 – the DIAMONDS approach injects invalid sequences of messages into the SUT. This kind of fuzzing is referred to as behavioural fuzzing. When behaviour fuzzing, the input data of the messages remains disregarded, only the order of messages is considered. The goal is to find security relevant weaknesses that consist rather in errors of the implemented state machine than in processing invalid input data. To a minor degree this has already been done by using state machines where invalid transitions are taken to determine the next message to be sent [Be10, HSD08]. This is rather random fuzzing than smart fuzzing because only invalid transitions are considered independently of their special meaning.

An example of a vulnerability that can be found only by behaviour fuzzing is given in [TMK10]. In a regular HTTP 1.1 GET request, two messages are sent to the web server: a GET message, that contains the requested document, and a host message that contains the domain name of the host that is requested. Earlier versions of Apache web server are vulnerable for a denial-of-service attack that consists in repeating the host message of a request. Repeating the host message can result in a crash of the web server. This vulnerability cannot be found using data fuzzing.

A more sophisticated kind of behaviour fuzzing can be done using UML sequence diagrams. Since UML 2, sequence diagrams can contain control structures like combined fragments, e.g. for branches and loops, that are guarded by constraints. By breaking these constraints, invalid behaviour can be triggered in a more model-based manner by evaluating more information of the model than just the order of states or messages. Moreover, different sequence diagrams can be merged starting with a valid sequence of messages representing one functional behaviour and ending with another.

3.4 Test Execution

Test execution is the phase whereby test cases obtained with the risk-based model-based security testing techniques proposed by the DIAMONDS project are actually executed against an SUT. Therefore this phase takes as input the results of the test generation phase, which may be automatically executable test scripts, manually executable test scenarios, or a combination of both. As depicted in Figure 2 the output from the test execution phase may consist of new weaknesses and vulnerabilities, which in turn would

provide input for a new iteration of the process, starting all over again from the risk analysis phase. An important benefit of the risk-based model-based security testing with regard to test execution is the fact that it facilitates the traceability of test results back to elements of risk analysis (threats, vulnerabilities, weaknesses, treatments …) they originated from, as well as to elements of system design potentially affected by the identified risks. This form of feedback will eventually have a positive impact on the overall security of the SUT.

4 Case Study

For this research project a connectivity module was chosen to represent the automotive domain. The so called connectivity module is used in vehicles to interconnect user electronic devices with the car's own infotainment system and in-vehicle network via Bluetooth and USB. The connectivity module is connected via CAN to the vehicular network as shown in Figure 3.

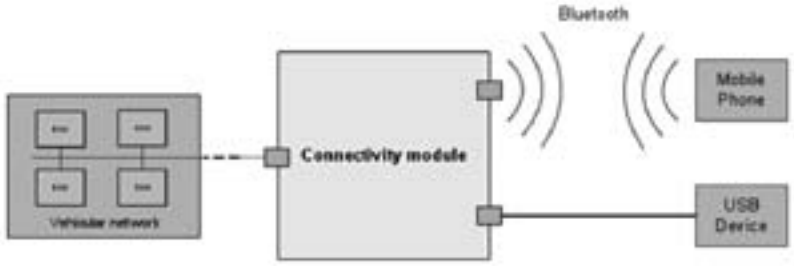


Figure 3: Connectivity module as used in the case study of Dornier Consulting

For the case study an existing test environment is used. This test environment provides a physical set-up of selected dashboard components of an automotive OEM. The behaviour of the set-up is described with a UML model, which basically consists of different sequence diagrams. The main use case which is being investigated in more detail in this case study is the connection of a Bluetooth device to the car's connectivity and communication module and the usage of several Bluetooth-based services.

The UML-based system model of the SUT is supplemented with a dedicated UML profile that was tailored to address the needs of the automotive domain. It is called Automotive Domain Modelling Language (ADML) and is part of Dornier Consulting's testing framework do.ATOMS. Most of the SUTs that are tested with do.ATOMS are reactive embedded real-time systems, e.g. head units, on-board-units and others which are relevant in the automotive or telecommunication domain. The test case design and execution supports modelling parallel, loop, and conditional activities, where discrete and continuous signals like brake/acceleration-sequences, speech recognition, tone detection, GSM calls, GPS-tracking, voltage-ramps, and other scenarios can be tested under real-time conditions.

The do.ATOMS framework

The name “do.ATOMS” stands for Automated Testing of Model Based System Design. Following this approach the framework serves as a MBT environment for discrete interconnected embedded systems. Test cases are generated from a given system model. The system model is based on UML and specifies the structure as well as the behaviour of the SUT. The system model gets processed by one of several model parsers – depending on the UML tool used for creating the system model (e.g. Sparx Systems Enterprise Architect or IBM Rational Rhapsody). The test cases can be transferred into a test case library and can be edited in an easy to use editor (see Figure 4) that shows the transformed test cases as workflows. During test case creation the system model parser also parameterize the test cases so that they are directly executable on the corresponding SUT test setup. The test case framework also handles test case management, execution, and reporting as well. There are also interfaces to support external test management solutions like HP Quality Center. The execution layer of do.ATOMS has a direct access to the interfaces provided by the SUT and follows a service-orientated approach. This gives do.ATOMS the flexibility to support a wide variety of different communication, data exchange, and functional interfaces which also simplifies the integration into the SUT environment.

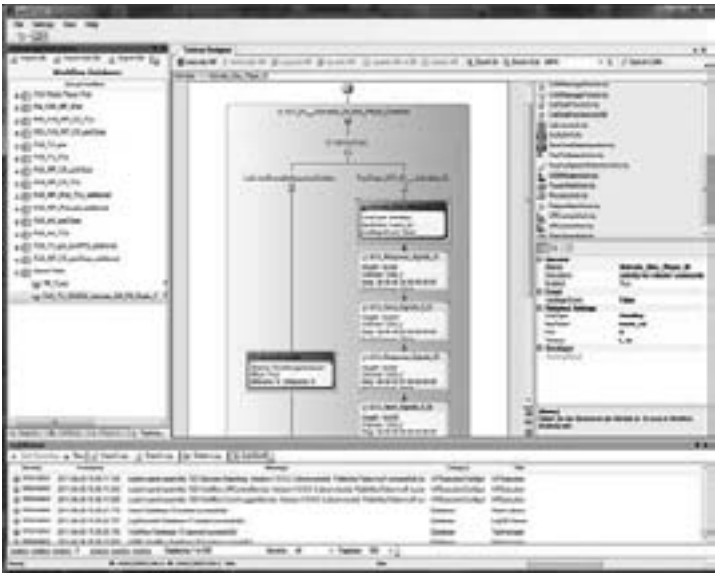


Figure 4: The do.ATOMS interface

Using do.ATOMS as a stable and flexible base Dornier Consulting has developed several concrete products on top of it. These products play a vital role in the realization of the company’s projects and are also used by its customers. One product example is do.ARTS (Automatic Robot Test System), a robot controlled test bench solution for

automated and model-based testing of complex Human-Machine Interface (HMI) functionalities.

In the progress of the DIAMONDS project the following aspects of do.ATOMS are being extended or enhanced:

- Support for new approaches and methodologies that allow the specification, generation and execution of IT security related tests in an automotive context (e.g. risk analysis and modelling).
- System modelling techniques get extended to allow the modelling of security requirements and features of the SUT.
- The model parser components will be able to convert the security requirements of the system model into corresponding security relevant test cases.
- Test case execution gets extended to be able to analyse and to verify the conformity of the SUT's security requirements.

4.1 Risk Analysis

The test approach of Dornier Consulting is based on a risk analysis made in CORAS as mentioned in Section 3.1. Each diagram in CORAS represents a threat scenario, which can be reached by using several different vulnerabilities. These vulnerabilities are applied with a value from 0.0 to 1.0, where 0.0 represents no protection and 1.0 represents full protection.

For each vulnerability an own connection from the threat to the threat scenario is defined. The connection is of type ‘initiates’ and is applied with a likelihood. According to the CORAS language specification the ‘threat scenario’ leads to an ‘unwanted incident’, which impacts an ‘asset’. All connections are applied with a likelihood-value, just like the ‘initiates’ connection. In Figure 5 the threat scenario ‘DoS without Pairing’ is shown as an example.

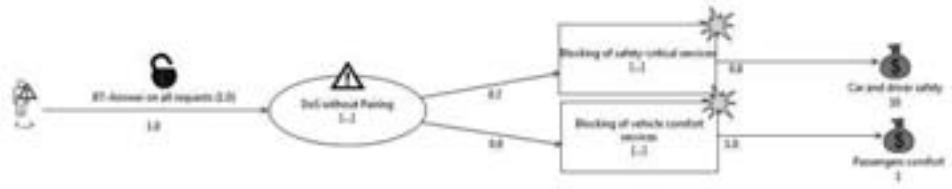


Figure 5: CORAS model for the threat scenario 'DoS without Pairing'

For this approach the used terms have the following meaning:

Likelihood Represents the likelihood that an event happens, which is expressed by the connection.

Vulnerability By vulnerability a flaw in the system is addressed. The value at the vulnerability represents the degree of protection. The main purpose of which to prevent such an event. Therefore 1.0 means full protection and 0.0 means no protection at all.

Consequence The consequence is applied to an asset. The value of the consequence represents the dimension of the consequence with respect to security. The higher the values, the stronger is the security impact.

The elements shown in Figure 6 are used in a CORAS diagram. The diagram usually begins with a **threat**, which has an 'initiate' connection to a **threat scenario**. From here a 'leads to' connects one or several **unwanted incidents**. The **unwanted incidents** have an 'impact' connection to the assets.

Figure 6: CORAS legend of used elements

4.2 Test Model Design

In order to cover the complete consequence of a vulnerability, it is necessary to iterate over each path of the CORAS diagram. The following equation for the calculation is used. N represents the number of possible paths from the threat scenario to the asset.

$$consequence = \sum_{i=1}^{N} (likelihood_{LeadsTo_i} \times likelihood_{Impacts_i} \times consequence_i)$$

In the example of Figure 5 the consequence would be 6.5, which was calculated by summing up both paths:

$$0.7 \times 0.8 \times 10.0 + 0.9 \times 1.0 \times 1.0 = 6.5$$

Based on the values for the vulnerability, likelihood, and consequence a score can be calculated. For this approach the following equation is used:

$$security\ score = \frac{(likelihood \times consequence)^2}{vulnerability}$$

This is a simple approach to categorize the different vulnerabilities. In this approach the highest security score does not represent the most relevant message, but indicates a general relevant message in respect to security.

In the example of Figure 5 the likelihood is 1.0, the vulnerability 1.0, and the consequence 6.5 as calculated in the previous paragraph. After putting these values into the presented equation we get a value of **42.25**.

To generate a sufficient set of test cases based on the system model, it is necessary to map the calculated security score to the system model. The most uncompromising way would be to model each path of CORAS to the system model. But this is not practical, since some threat scenarios are too complicated to model (e.g. exploits). Therefore the approach requires a scenario for each vulnerability to be modelled. These 'CORAS scenarios' describe how to reach certain vulnerabilities and are added in a separate package to the system model. It should be noticed that with this modification the system model becomes a test model.

In the sequence diagram 'functions' are modelled to represent the messages sent between parts. The messages used in the 'CORAS scenario', which are derived from the CORAS analysis, have the same identifier as the messages used in the system model. Therefore the security score of the messages of the 'CORAS scenarios' can be mapped to a single message of the system model. If a message is used in several 'CORAS scenarios' the security score will be aggregated to a single value. With this approach messages of the global system model can be applied with a security score.

For instance all the messages, which lead to the threat scenario 'DoS without Pairing', will get a value of 42.25. If one of these messages is used in another CORAS scenario as well, the value of the other CORAS scenarios will be added to the 42.25.

For testing purpose the security score is used to calculate a priority of the different messages. Together with the occurrence of a single message in the system model a priority is calculated which indicates the testing framework the relevant messages with respect to security. This priority influences the test case generation.

The presented approach allows mapping the CORAS risk analysis to the system model. The introduced equation for the security score is a simple way to establish a security ranking between the different vulnerabilities of the CORAS risk analysis. Based on the analysis the scenarios can be extracted to the system model. The containing messages can then aggregate the security scores of the CORAS scenarios, in which they are included. In combination to the occurrence of a message within the system model a priority order can be introduced with respect to security.

4.3 Test Generation

As mentioned in the do.ATOMS overview, during test case generation the system model gets analysed by one of the framework's model parsers and is then transformed into a do.ATOMS internal representation, the so-called workflow. These workflows in turn can also be considered as an independent test model, which is – in this case – automatically derived from the system model. This step is therefore also called "model-to-model-transformation". Each workflow consists of several workflow activities that interact with the SUT (e.g. via sending a CAN or Bluetooth message) or that monitor its state (verification of the presence of certain CAN messages and the like). The sequence of the activities can also be altered in various ways using workflow flow control constructs (like if/else branches or loops). All test cases follow the uniform approach that do.ATOMS

simulates the external Bluetooth end user device to test the Bluetooth connectivity features of the connectivity module.

During the transformation phase security relevant information (like the security score metadata) that is part of the system model is also transferred and is still present in the do.ATOMS workflow model. This allows the application of security testing techniques like data fuzzing.

Each derived test case (workflow) that is transferred into the test case library of the do.ATOMS test case designer can be transformed into a fuzzing test case. A fuzzing test case can be seen as a "malicious" counterpart of a test case that represents the normal system behaviour. As mentioned before, the workflow activities are responsible for sending stimuli to the SUT.

For finding entry points for data fuzzing in these activities the following approach was chosen: Each activity of the test cases is examined and its security score information is evaluated. If the security score is above a configurable threshold, the activity is enclosed by a so-called fuzzing container. This element is responsible for injecting the fuzzing functionality by manipulating its child element e.g. the payload of a message. This can be done at design time when the workflow is created or at runtime when the workflow gets executed.

Figure 7 shows an excerpt of a fuzzing workflow that handles the Bluetooth pairing process (e.g. between the connectivity module and the driver's mobile phone). The data fuzzing container has been applied to "EnterPIN_I4" activity, which in this example has the highest security score with a value of 532.

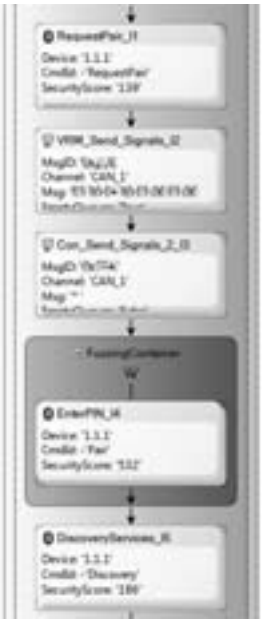


Figure 7: An excerpt of a fuzzing workflow that handles the Bluetooth pairing process

To sum up, the following figure shows the complete conceptual approach for test case generation that is realized with do.ATOMS:

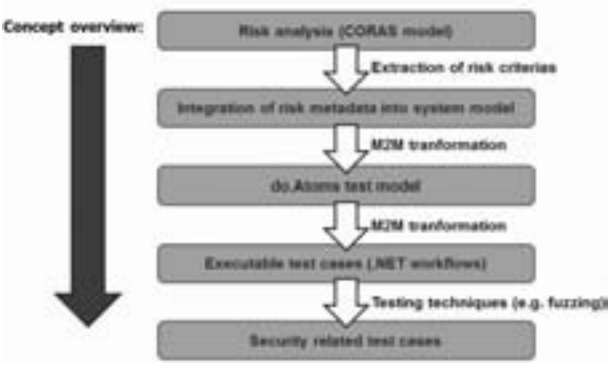


Figure 8: An overview of the case study approach

4.4 Test Execution

In general, the execution of a fuzzing test case is somewhat different compared to the execution of a normal test case. What differs the most is the process of determining the test result. When a functional test case is finished the SUT can be examined is some way

(e.g. by inspecting the state of the SUT) in order to find out whether the test was successful or not. When it comes to the execution of fuzzing test cases the situation changes since the determination of the test result is often not clear. For each fuzzing test case one has to find measurable criteria for deriving the test result, e.g. the SUT could ignore a fuzzing attack completely and just continue to function normally or it faces a fatal crash.

In terms of do.ATOMS-based test execution the workflow itself defines the test case control flow, so each activity (test step) is executed in the pre-defined order and triggers a certain test step functionality (like sending or validating CAN or Bluetooth messages to the SUT or triggering an image-based validation of the head unit display). This is also the case for fuzzing test cases – the main difference is the fuzzing container which manipulates its child activity to apply data fuzzing. The particular manner in which this data fuzzing is applied depends on the action performed by the child activity.

Concerning data fuzzing, the creation of semi-valid input is of particular importance. In general, how this input is generated depends on the specific protocol that is fuzzed. Taking the example of sending a PIN to the connectivity module the fuzzing container investigates the child activity and identifies the parameter, which represents the PIN. It inspects the data type of the parameter, which in this case is specified as “string”, consisting of at least a 4-digit number. Depending on the data type various fuzzing methods are applied by the fuzzing container to create a semi-valid version of the expected input data (like sending characters instead of digits or a PIN with an unexpected length, e.g. consisting of 1000 random digits). Additionally it can also be helpful to use invalid characters that have special meaning, for instance null characters that terminate C strings, or the percentage character that is used for formatted strings. While a dumb fuzzer would generate invalid input data for most of the messages, a model-based fuzzer has enough knowledge to fuzz only the parameters of a certain activity, e.g. the activity sending the message that carries the PIN. The fuzzing container itself can also be configured in various ways to control its runtime behaviour. One example is that the container can be instructed to create a new fuzzed input value for each test run.

This fuzzing approach is realized by implementing the fuzzing library provided by Fraunhofer FOKUS. This library encapsulates all fuzzing related business logic and allows do.ATOMS the retrieval of fuzzed input data values by simply sending an XML request. This request contains a definition of the data type or structure that should be fuzzed and the fuzzing generators that should be applied to it. The fuzzing library will also be capable of supporting behavioural fuzzing which will be applied to the Dornier case study later on as well. Behaviour Fuzzing will affect the order in which the workflow activities get executed.

5 Conclusion

Increasing complexity in ICT systems and automotive electronics as well as the increased integration of vehicle networks with ICT networks lead to increased IT security risks for the consumer. This trend and the accelerating development cycles for new

products, features, and applications in this area require efficient and fast development and integration methods and tools.

One important approach is to transfer model-based development concepts to the test and integration phases by introducing MBT. Due to the mentioned increase of security issues in ICT and automotive electronics, within the European research project DIAMONDS the MBT approach is currently extended to security focused testing.

Within DIAMONDS the authors developed, together with project partners from six European countries, a new method and process on how to efficiently generate security focused test scenarios and test cases.

This new risk-based methodology has been applied to identify possible flaws in the implementation of a vehicle infotainment system by analysing its UML system model. Based on this risk analysis a security related prioritization of the different Bluetooth messages can be achieved which enables a smart and efficient generation of the most critical and relevant test scenarios. Adapted security test generation methods based on known concepts such as fuzzing of these pre-defined test scenarios will then be used to generate the final set of test cases.

This paper discussed the generation process for risk based security test cases using the model-based testing approach, which focus on the analysed security flaws. An approach that refers to the generation of test cases that cover possible unknown flaws by integrating data and behaviour fuzzing methodologies is also be examined.

The practical implementation of this generation process has been demonstrated by the extended MBT frame work do.ATOMS from Dornier Consulting for industrial and commercial use and application.

Within the next months the developed methodology will be concretized, optimized, and validated through the automotive case study to demonstrate the efficiency, robustness, and usability based on the implementation in the do.ATOMS test design and execution frame-work.

6 References

[Am00] Amland, S.: Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. J. of Systems and Software 53 (3), 287-295 (2000)

[Be10] Becker, S., Abdelnur, H., State, R., Engel, T.: An Autonomic Testing Framework for IPv6 Configuration Protocols. In: Proceedings of the Mechanisms for autonomous management of networks and services (AIMS'10), pp. 65-76. Springer (2010)

[Be11] Bekrar, S.; Bekrar, C.; Groz, R.; Mounier, L.: Finding Software Vulnerabilities by Smart Fuzzing. In: Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, pp. 427-430. IEEE Computer Society, Washington, D.C. (2011)

[GOR07] Güdemann, M., Ortmeier, F., Reif, W.: Using Deductive Cause Consequence Analysis (DCCA) with SCADE. In: Saglietti, F., Oster. N. (eds.): Proceedings of SAFECOMP

2007, LNCS, vol. 4680, pp. 465-478. Springer (2007) [He03] Herzog, P.: OSSTMM 2.1. Open-Source Security Testing Methodology Manual. Institute for Security and Open Methodologies (2003)

[HSD08] Hsu, Y., Shu, G., Lee, D.: A Model-based Approach to Security Flaw Detection of Network Protocol Implementations. In: Proceedings of IEEE International Conference on Network Protocols (ICNP 2008), pp. 114-123. IEEE (2008)

[Ko10] Koscher, K., Checkoway, S., et al.: Experimental Security Analysis of a Modern Automobile. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P'10), pp. 447-462 (2010)

[Ko11] Koscher, K., Checkoway, S., et al.: Comprehensive Experimental Analyses of Automotive Attack Surfaces. In: Proceedings of the 20th USENIX conference on Security. USENIX Association, Berkeley (2011)

[LSS11] Lund, M.S., Solhaug, B., Stlen, K.: Model-Driven Risk Analysis. The CORAS Approach. Springer (2011)

[MTL09] Murthy, K. K., Thakkar, K. R., Laxminarayan, S.: Leveraging Risk Based Testing in Enterprise Systems Security Validation. In: Proceedings of the First International Conference on Emerging Network Intelligence (EMERGING 2009), pp. 111-116 (2009)

[Oe05] Oehlert, P.: Violating Assumptions with Fuzzing. IEEE Security & Privacy, vol. 3, no. 2, pp. 58-62. IEEE Computer Society, Los Alamitos (Mar.-Apr. 2005)

[Re04] Redmill, F.: Exploring Risk-based Testing and Its Implications: Research Articles. J. Softw. Test. Verif. Reliab. 14 (1), 3-15 (Mar. 2004)

[Re05] Redmill, F. 2005. Theory and practice of risk-based testing: Research Articles. J. Softw. Test. Verif. Reliab. 15 (1), 3-20 (Mar. 2005)

[SMP08] Stallbaum, H., Metzger, A., Pohl, K.: An Automated Technique for Risk-based Test Case Generation and Prioritization. In: Proceedings of the 3rd international workshop on Automation of software test (AST'08), pp. 67-70. ACM New York (2008)

[Ta09] Takanen, A.: Fuzzing: the Past, the Present and the Future. Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC), pp. 202-212 (2009)

[TDM08] Takanen, A., DeMott, J., Miller. C.: Fuzzing for Software Security Testing and Qualitiy Assurance. Artech House, Boston (2008)

[TMK10] Takahisa, K., Miyuki, H., Kenji, K.: AspFuzz: A State-aware Protocol Fuzzer based on Application-layer Protocols. In: Proceedings of Conference on IEEE Symposium on Computers and Communications (ISCC 2010), pp.202-208 (2010)

[Ze11] Zech, P.: Risk-Based Security Testing in Cloud Computing Environments. PhD Symposium at the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011), pp.411-414 (2011)

[Zi09] Zimmermann, F., Eschbach, R., Kloos, J., Bauer, T.: Risk-based Statistical Testing: A Refinement-based Approach to the Reliability Analysis of Safety-Critical Systems. In: Proceedings of 12th European Workshop on Dependable Computing (2009)

ISO 26262 – Quo vadis?

Stefan Kriso

Center of Competence „Functional Safety“
Robert Bosch GmbH
CoC-FS, CC/EVD
Robert-Bosch-Allee 1
74232 Abstatt
stefan.kriso@de.bosch.com

Abstract: Die ISO 26262 ist veröffentlicht und trägt damit einerseits zum Stand der Technik bei der Entwicklung sicherer elektrischer/elektronischer Systeme im Automobil bei. Andererseits sind einige Anforderungen der ISO 26262 visionär formuliert, so dass es nicht möglich ist, die Norm zu ihrem Veröffentlichungszeitpunkt umgesetzt zu haben, sondern es ist ein Einführungszeitraum notwendig. In dieser Zeit ist die Branche aufgefordert, praktikable und umsetzbare Interpretation der ISO 26262 zu finden, die der Intention der Norm möglichst nahe kommen sowie die Produktsicherheit nicht gefährden. Der Artikel nennt hierfür Beispiele und mögliche Lösungen.

1 Situationsbeschreibung

Die ISO 26262 wurde im November 2012 veröffentlicht und trägt damit zum Stand der Technik bei der Entwicklung sicherheitsrelevanter elektrischer/elektronischer Systeme in Straßenfahrzeugen bei. Ihre Umsetzung ist zur Reduktion unberechenbarer Produkthaftungsrisiken dringend angeraten [KS09] und sie steht deshalb derzeit im Fokus der Automobilbranche wie kaum eine andere Norm zuvor.

Neben dieser formaljuristischen Sicht auf die ISO 26262 gibt es allerdings noch die technisch-inhaltliche: Ein Teil ihrer Anforderungen wurde – teils bewusst, teils unbewusst – abstrakt und visionär formuliert. Dies bedingt, dass eine vollständige Umsetzung der ISO 26262 zu ihrem Veröffentlichungstermin nicht abgeschlossen sein kann, sondern dass ein Einführungszeitraum notwendig ist, während dessen die Anforderungen der Norm interpretiert und auf die tatsächlich vorhandenen Gegebenheiten in den jeweiligen Organisationen angepasst werden müssen [Re11].

Unterschiedliche Interpretationen verschiedener Beteiligter (OEM, Zulieferer, …) führen insbesondere an deren Schnittstellen zu Inkompatibilitäten, welche zu Reibungsverlusten führen, die der Produktsicherheit eher ab- als zuträglich sind. Es sollten daher gemeinsame Interpretationen der ISO 26262 gefunden werden, die einerseits der

Produktsicherheit dienen, andererseits aber Wettbewerbsdifferenzierung ermöglichen und Innovationsfähigkeit erhalten [KHK11, KH11].

Bis dahin gilt es, die abstrakten und visionären Anforderungen der ISO 26262 zu benennen und entsprechend ihrer Intention umzusetzen. Im Folgenden werden hierfür Beispiele genannt und Lösungsmöglichkeiten aufgezeigt.

2 ASIL-Einstufung

Die Vorgehensweise zur Ermittlung der ASIL-Einstufung einer Fehlfunktion eines E/E-Systems ist durch die ISO 26262 vorgegeben. Nach Bestimmung der Parameter für die Häufigkeit der Situation (Exposure „E"), der Beherrschbarkeit der Fehlfunktion (Controllability „C") und der Schwere der Auswirkung (Severity „S") ergibt sich der ASIL gemäß folgender Tabelle:

Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Abbildung 1: Tabelle zur Ermittlung des ASIL aus den Parametern S, E, C [ISO11].

Diese Vorgehensweise erscheint nur auf den ersten Blick objektiv. Die Ermittlung der Parameter E, C und S ist äußerst subjektiv, wie folgendes Beispiel zeigt: Versuchen zwei Personen die Häufigkeit der Situation „Bergabfahrt" zu ermitteln, von denen die eine aus dem norddeutschen Flachland kommt, die andere aber aus den Südtiroler Alpen, so werden sie möglicherweise auf Grund ihres persönlichen Erfahrungshintergrunds zu signifikant unterschiedlichen E-Einstufungen kommen. Wie aus Abbildung 1 ersichtlich ist, kann eine Varianz im Parameter E (wie auch bei den anderen beiden Parametern) direkt auf den ASIL durchschlagen: beispielsweise würde eine Variation in E von E1 bis E3 eine Variation im ASIL von A bis C bewirken (bei C3 und S3). Vor diesem Hintergrund ist es nicht verwunderlich, wenn dieselbe Fehlfunktion von unterschiedlichen Beteiligten mit unterschiedlichem ASIL eingestuft wird. Verstärkt

wird dies durch eine momentane Unsicherheit in der Ermittlung der E-, C- und S-Parameter, da die ISO 26262 sich hierzu im normativen Teil nicht äußert und sich daher nach und nach verschiedene Vorgehensweisen in der Branche herausbilden (z.B. rein qualitative Betrachtungen vs. Durchführung von Fahrversuchen). Diese unterschiedlichen Vorgehensweisen sind zwar im Allgemeinen jede für sich normkonform und sinnvoll, sie führen jedoch teilweise zu unterschiedlichen und nicht vergleichbaren Ergebnissen. Insgesamt ist dies nicht ohne Konsequenzen:

- Hat sich in der Branche bezüglich einer bestimmen Fehlfunktion eine ASIL-Einstufung weitgehend etabliert und es weicht ein einzelner Hersteller mit seiner ASIL-Einstufung von diesem impliziten Branchenkonsens nach unten ab (d.h. er stuft die Fehlfunktion geringer, möglicherweise gar als QM ein) und es kommt gerade bei diesem Hersteller zu einem Produkthaftungsfall, so wird er möglicherweise dem Vorwurf ausgesetzt sein, er hätte die geringere Einstufung aus Gründen der Kosteneinsparung vorgenommen. Weiter könnte ihm vorgeworfen werden, den Stand der Technik nicht berücksichtigt zu haben, denn hätte er sich an den Einstufungen der anderen Hersteller orientiert und sein Produkt dementsprechend entwickelt, so wäre der Unfall möglicherweise vermeidbar gewesen.

- Der umgekehrte Fall, dass ein Hersteller sein Produkt bzw. die entsprechende Fehlfunktion höher einstuft als der Rest der Branche, scheint zunächst unkritisch. Er betreibt in diesem Fall mehr Entwicklungsaufwand als es nach Branchenmeinung offensichtlich notwendig wäre, und das Produkt wird möglicherweise teurer, aber nicht unbedingt sicherer. Unter Umständen wird aber durch die höhere Einstufung ein neuer Stand der Technik geschaffen, den dann auch die anderen umsetzen müssen, um nicht die Verkehrsfähigkeit ihrer Produkte zu gefährden. Die Folge der Höhereinstufung eines Einzelnen könnte also in diesem Fall zu einem Pushen der ASIL-Einstufung in der gesamten Branche nach oben führen, ohne dass es notwendigerweise zu einem Sicherheitsgewinn kommt. Als weitere Folge könnte dies mittel- bis langfristig zu einem „Hochschaukeln“ der ASIL-Einstufungen führen, womit im Extremfall die Idee des ASILs ad absurdum geführt wäre; das Ziel und der Benefit der ASIL-Einstufungen, nämlich den Entwicklungsaufwand an Hand der Sicherheitsrelevanz des Produkts zu skalieren, wäre weniger stark bis gar nicht mehr gegeben.

- Insbesondere für einen Zulieferer ergibt sich die zusätzliche Schwierigkeit, dass er von seinen Kunden für dieselbe Fehlfunktion mit unterschiedlichen ASIL-Einstufungen konfrontiert wird. Möchte er mit einer Plattformentwicklung aber mehrere Kunden bedienen, stellt sich für ihn die Frage, nach welchem ASIL er diese entwickelt. Orientiert er sich am höchsten ASIL, steckt er möglicherweise für diejenigen Kunden, die einen geringeren ASIL fordern, mehr Aufwand in die Entwicklung als der Kunde bereit ist zu zahlen. Entwickelt er aber nach dem

geringeren ASIL[1], wird er diejenigen Kunden, die einen höheren ASIL fordern, ohne weitere Zusatzmaßnahmen nicht bedienen können.

Aus vorgenannten Gründen liegt es im Brancheninteresse, ein einheitliches Bild über die ASIL-Einstufungen verschiedener Fahrzeugfunktionen zu bekommen. Zumindest für die rudimentären Grundfunktionen sollten fahrzeugunabhängige ASIL-Einstufungen nach einer einheitlichen Methodik vorgenommen werden, um zum einen ein einheitliches „ASIL-Gefüge" zu bekommen, um zum anderen aber auch eine Methodik bereitzustellen, nach der neue Fahrzeugfunktionen klassifiziert werden können, die dann ebenfalls in dieses ASIL-Gefüge passen.

Selbstverständlich ist die letztendliche ASIL-Einstufung fahrzeugabhängig und kann (und wird) von einem Katalog harmonisierter ASIL-Einstufungen abweichen. Dennoch kann ein solcher in einem ersten Schritt als Orientierungshilfe zur Einstufung neuer Systeme dienen.

Die Anforderungen und Beispiele in der ISO 26262 orientieren sich im Wesentlichen an Personenkraftwagen bis 3500 kg zul. Gesamtmasse. Dementsprechend wurde auch der Scope der ISO 26262 formuliert: „*ISO 26262 is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3 500 kg*." [ISO11]. Es herrscht jedoch Branchenkonsens, dass die ISO 26262 grundsätzlich auch auf andere Fahrzeugklassen, z.B. LKW oder Motorräder, angewendet werden kann und soll[2]. Unsicherheit besteht lediglich noch bei einigen Anforderungen in der Art und Weise, wie diese bei den anderen Fahrzeugklassen umzusetzen sind, so z.B. auch bei der Methodik der ASIL-Einstufung: Geht man beispielsweise bei der ASIL-Einstufung implizit von mehrspurigen Fahrzeugen aus, so mögen bestimmte Fehlfunktionen relativ einfach beherrschbar sein oder eine geringe Auswirkungsschwere haben. Bei einem Motorrad dagegen kann dasselbe System unter Umständen eine deutlich höhere ASIL-Einstufung bekommen, weil beispielsweise die Beherrschbarkeit schlechter oder die Auswirkungsschwere höher ist. Es ist also darauf zu achten und es sollte auch dementsprechend in der Beschreibung der Methodik berücksichtigt werden, dass durch Erweiterung auf die übrigen Fahrzeugklassen einerseits keine Inkonsistenzen entstehen, andererseits die Spezifika der unterschiedlichen Fahrzeugklassen ausreichend und angemessen berücksichtigt werden. Vordringliche Aufgabe für die nächste Überarbeitung der ISO 26262 ist daher nicht nur die Berücksichtigung der übrigen Fahrzeugklassen im Scope der Norm, auch in den Anforderungen an sich müssen diese adäquat Berücksichtigung finden.

[1] Hier sei angenommen, dass eine geringere Einstufung des Kunden dadurch gerechtfertigt ist, weil er in seinem System, das außerhalb des Fokus des Zulieferers liegt, weitere Maßnahmen implementiert hat, die auf Gesamtsystemebene einen höheren ASIL ergeben (ASIL-Dekomposition). Andernfalls sollte der Zulieferer abwägen, ob eine Entwicklung nach dem höheren ASIL nicht doch gerechtfertigt ist.

[2] Siehe z.B. [BMW12]: „Erstmals bei einem Zweirad mit elektrischem Antrieb kommen hier die von den führenden Automobilherstellern erarbeiteten Standards für Hochvoltsicherheit (> 60 Volt Gleichspannung) und Funktionssicherheit zum Tragen. Die Entwicklung gemäß ISO 26262 ist bei (Elektro-) Zweirädern bisher einmalig und stellt sicher, dass sämtliche funktionsrelevanten Umfänge normgerecht und nach dem Stand von Wissenschaft und Technik entwickelt werden."

3 Qualifikation von Software-Werkzeugen

An vielen Stellen im Entwicklungsprozess elektrischer/elektronischer Systeme kommen Software-Werkzeuge zum Einsatz. Hiermit sind alle Werkzeuge gemeint, die in Software realisiert sind – nicht nur diejenigen, die in der Software-Entwicklung eingesetzt werden, sondern überall bei der Entwicklung elektrischer/elektronischer Systeme. Neben Werkzeugen für die Software-Entwicklung schließt dies zum Beispiel Anwendungen, die bei der Hardware-Entwicklung, der Steuergerätekalibrierung oder auf Systemebene, z. B. für das Anforderungsmanagement eingesetzt werden, mit ein. Das Ziel der Qualifikation von Software Tools ist es, durch Absicherung der Werkzeuge und deren geeignete Verwendung Fehler, die durch den Einsatz dieser Tools in das Produkt gelangen könnten, zu verhindern. Hierfür sieht die Norm folgende Fallunterscheidung vor: Einerseits kann, zum Beispiel durch entsprechende Maßnahmen in der Entwicklung, vermieden werden, dass das Werkzeug selbst Fehler erzeugt, andererseits können durch das Werkzeug erzeugte Fehler durch geeignete Absicherungsschritte (z.B. Verifikation, Test, Review) in einem nachgelagerten Prozessschritt oder durch organisatorische Maßnahmen identifiziert und behoben werden. Um mit der ISO 26262 konform zu sein, müssen alle Software-Werkzeuge, die bei der Entwicklung eines sicherheitsrelevanten elektronischen Produkts zum Einsatz kommen, im Kontext des spezifischen Entwicklungsprozesses dieses Produkts betrachtet werden. Eine generelle „ISO 26262-Konformität“ von Werkzeugen unabhängig vom jeweiligen Anwendungsfall ist gemäß der Norm nicht möglich – auch die häufig verwendete Qualifikation in so genannten „Referenzprozessen“ erfüllt dieses Ziel für den spezifischen Entwicklungsprozess eines Herstellers nicht; es muss immer ein Abgleich des Referenzprozesses mit dem tatsächlich vorhandenen spezifischen Entwicklungsprozess stattfinden. Jedoch kann aus Gründen der Wiederverwendung von Ergebnissen eine Vorabqualifikation von Werkzeugen für Standardanwendungsfälle sinnvoll sein, wenn sich deren Einsatz im Entwicklungsprozess aus einer Standardkonfiguration ableiten lässt. Sie kann beispielsweise die Zertifizierung eines Tools im Rahmen eines ISO 26262-konformen Prozesses bei richtiger Auslegung deutlich erleichtern.

Die ISO 26262 beschreibt in Teil 8 Kapitel 11 „Confidence in the use of software tools“ die Aktivitäten, die hier durchzuführen sind:

Bei der Qualifikation werden im ersten Schritt alle Anwendungsfälle, bei denen ein Werkzeug zum Einsatz kommt, klassifiziert. Zum Beispiel: Als nicht sicherheitsrelevant wird eingestuft, wenn ein Textverarbeitungsprogramm, das zur Dokumentation verwendet wird, beim Abspeichern die Schriftart verändert. Anders stellt sich der Fall dar, wenn beim Abspeichern relevante Informationen verfälscht werden. Im ersten Fall beeinflusst das Werkzeug den Inhalt des Ergebnisses nicht und es sind keine weiteren Qualifizierungsmaßnahmen notwendig. Im zweiten Fall muss geklärt werden, mit welcher Wahrscheinlichkeit in einem späteren Schritt im Entwicklungsprozess fehlerhafte Inhalte, die vom Werkzeug ausgegeben werden, zum Beispiel durch Reviews oder Tests entdeckt werden können.

Absicherungsmaßnahmen lassen sich durch ein weiteres Werkzeug oder einen überlagerten Prozessschritt (z. B. Plausibilisierung) vornehmen. Wenn Werkzeugfehler

durch geeignete Verifikationsschritte im Entwicklungsprozess nicht mit hoher Wahrscheinlichkeit ausgeschlossen werden können, sind Maßnahmen für die Werkzeugqualifizierung erforderlich. Das Ergebnis der Klassifizierung (siehe Abbildung 2) hängt wesentlich davon ab, in welchem Zusammenhang ein Software Tool im Entwicklungsprozess verwendet wird.

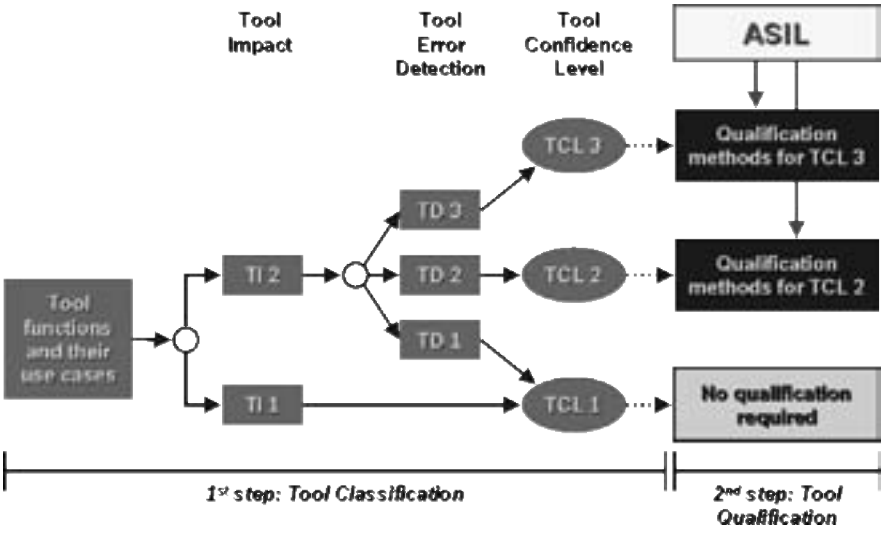


Abbildung 2: Ablauf bei der Qualifikation von Software-Werkzeugen nach ISO 26262.

Für die bei „TCL2“ oder „TCL3“ an die Klassifizierung anschließende Qualifizierung von Werkzeugen stehen nach ISO 26262 vier Methoden zur Auswahl („Increased confidence from use“, „Evaluation of the development process“, „Validation of the software tool“ und „Development in compliance with a safety standard“). Welche der alternativen Maßnahmen für eine Qualifizierung in Betracht kommen, wird durch die ASIL-Einstufung des zu entwickelnden Produkts und durch das Ergebnis der Werkzeugklassifizierung bestimmt. Abbildung 3 gibt eine Übersicht über die Qualifizierungsmethoden, die für Werkzeuge der Klasse „TCL 3“ für die verschiedenen ASIL empfohlen werden.

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Increased confidence from use in accordance with 11.4.7	++	++	+	+
1b	Evaluation of the tool development process in accordance with 11.4.8	++	++	+	+
1c	Validation of the software tool in accordance with 11.4.9	+	+	++	++
1d	Development in accordance with a safety standard*	+	+	++	++

* No safety standard is fully applicable to the development of software tools. Instead, a relevant subset of requirements of the safety standard can be selected.

EXAMPLE Development of the software tool in accordance with ISO 26262, IEC 61508 or RTCA DO-178.

Abbildung 3: Empfohlene Maßnahmen zur Qualifizierung von Software-Werkzeugen für TCL3 [ISO11].

Die Qualifikation von Software-Werkzeugen kann auf Grund der Menge der eingesetzten Werkzeuge sehr aufwändig werden. Bei Bosch sind z.B. in 56 Organisationseinheiten im Automobilbereich ca. 1500 Software-Werkzeuge im Einsatz. Bei einer systematischen Betrachtung wurden im ersten Schritt alle Werkzeuge klassifiziert. Hierbei zeigte sich, dass ca. 98% der Werkzeuge in TCL1 eingestuft werden können, da durch ihre Verwendung im Entwicklungsprozess bzw. durch ihre Einbettung in Werkzeugketten sichergestellt werden kann, dass fehlerhafte Ausgaben eines Werkzeugs in nachgelagerten Review-, Test- oder Plausibilisierungsschritten gefunden werden und sich somit nicht mehr sicherheitskritisch im Endprodukt manifestieren können. Nur für die restlichen 2% der Tools mit einer TCL2- oder TCL3-Einstufung musste ein aufwändigerer Qualifikationsschritt durchgeführt werden. Der im ersten Klassifikationsschritt entstandene Aufwand wurde also dadurch mehr als kompensiert, dass für den zweiten Schritt der Qualifikation in Summe deutlich reduzierte Aufwände entstanden sind.

4 Änderungen an bestehenden Systemen

Bezüglich Modifikationen erwähnt der Scope der ISO 26262: „*For further development or alterations based on systems and their components released for production prior to the publication date of ISO 26262, only the modifications will be developed in accordance with ISO 26262*.“ [ISO11]. In der Realität heißt das, dass bei einer Änderung lediglich die Änderung nach ISO 26262 durchgeführt wird. Insbesondere bei einer Änderung an einem schon bestehenden, vor Normveröffentlichung freigegebenen System führt dies nicht zwangsläufig zu einem „Reengineering“ des bestehenden, unveränderten Altsystems. Doch wie kann eine Änderung einerseits normkonform, andererseits aber auch effizient durchgeführt werden? Abbildung 4 zeigt in einem Ablaufdiagramm, wie bei einer Änderung an einem bestehenden System vorgegangen werden kann:

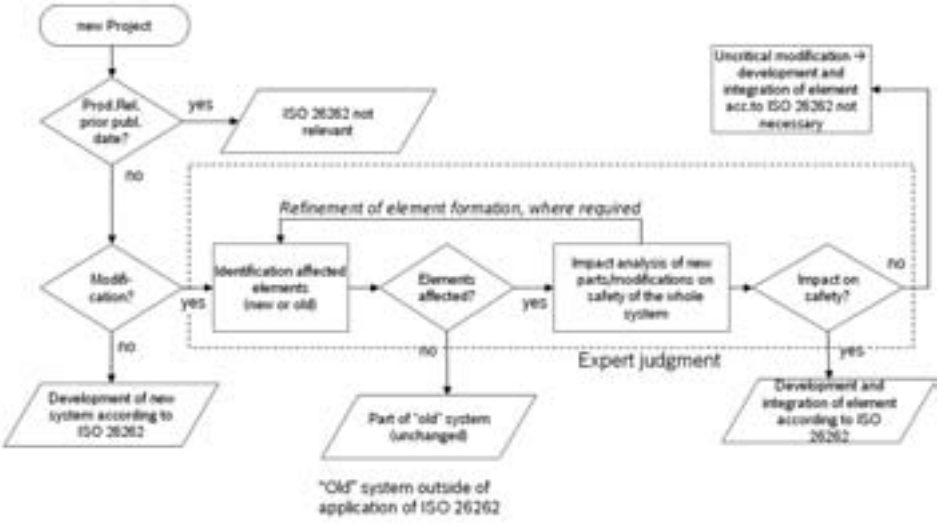


Abbildung 4: Ablauf bei einer Änderung an einem bestehenden Produkt.

Im ersten Schritt wird entschieden, ob die ISO 26262 überhaupt anzuwenden ist (Produktionsfreigabe nach Normveröffentlichung) und ob es sich um eine Änderung an einem bestehenden System handelt. In diesem Fall sind zunächst diejenigen Systemelemente zu identifizieren, die von der Änderung betroffen sind. Die nicht betroffenen Elemente sind Teil des unveränderten „Altsystems", welches nicht im Scope der ISO 26262 ist.

Für die von der Änderung betroffenen Systemelemente ist der Einfluss der Änderung auf das Gesamtsystem zu analysieren. Hierbei kann es sich durchaus ergeben, dass Systemelemente, die zunächst dem unveränderten „Altsystem" zugerechnet waren, sich doch als von der Änderung betroffen erweisen – somit sollte bei der Einflussanalyse von einer iterativen Vorgehensweise ausgegangen werden.

Kann durch eine Expertenbewertung gezeigt werden, dass die geplanten Änderungen definitiv keinen Einfluss auf die Sicherheit des Gesamtsystems haben, können diese Änderungen als unkritisch angesehen werden, und für die Entwicklung bzw. Integration der geänderten Anteile ist nicht unbedingt die formale Anwendung der ISO 26262 notwendig. Dies hätte für die Produktsicherheit keinerlei Mehrwert, d.h. keine Erhöhung der Systemsicherheit zur Folge.

Ein Beispiel für eine solche unkritische Änderung könnte eine Software-Parameteränderung sein. Bei neuen Parametern handelt es sich um eine Änderung, bei der rein formal die ISO 26262 angewendet werden müsste. Findet diese Parameteränderung in einer wohlbekannten Plattform statt und bewegen sich die Änderungen in bekannten Parametergrenzen anderer Plattformprojekte, so mag es möglich sein durch eine Expertenbewertung und durch Felderfahrungen bekannter Referenzprojekte zu zeigen, dass diese Parameteränderungen unkritisch sind und eine

formale Anwendung der ISO 26262 hier keinen Benefit bringt. Somit könnte in diesem Fall auf die formale Anwendung der ISO 26262 verzichtet werden.

5 Open Source Software und Tools

Bei Anwendung der ISO 26262 stellt sich die Frage, wie mit Open Source Software und Open Source Tools umgegangen werden muss und ob diese überhaupt bei der Entwicklung sicherheitsrelevanter Systeme eingesetzt werden dürfen. Die Anforderungen der ISO 26262 machen hier prinzipiell keine Ausnahme bezüglich Open Source bzw. stellen keine expliziten Anforderungen. Open Source Software ist daher prinzipiell genau so zu behandeln wie die übrige Software, für die Tools gilt dasselbe entsprechend:

- Open Source Codefragmente können verwendet werden, wenn deren Entwicklung gemäß ISO 26262-6 (Software development) abläuft.

- Open Source Softwarekomponenten können in einem nicht-sicherheitsrelevanten Software-Teil verwendet werden, wenn die „Freedom from interference“ (ISO 26262-6, Kap. 7, ISO 26262-9, Kap. 6) zum sicherheitsrelevanten Software-Teil gezeigt werden kann. Sie müssen qualifiziert werden (ISO 26262-8, Kap. 12), wenn der sicherheitsrelevante Software-Teil nicht weggekapselt werden kann. Voraussetzung ist, dass die Open Source Softwarekomponente gemäß eines „angemessenen“ nationalen/internationelen Standards (z.B. CMMI, Spice, …) entwickelt wurde.

- Open Source Tools können eingesetzt werden, wenn sie gemäß ISO 26262-8, Kap. 11 („Confidence in the use of software tools“) qualifiziert wurden (siehe oben).

Prinzipiell steht die ISO 26262 der Verwendung von Open Source Software nicht im Wege; die Anforderungen der ISO 26262 sind so formuliert, dass sie auch auf Open Source Software anwendbar sind. In wie weit diese in der Praxis insbesondere bei Open Source Codefragmenten und Softwarekomponenten umsetzbar sind, muss kritisch geprüft werden, insbesondere wenn die Konformität der Open Source Entwicklungsprozesse mit der ISO 26262 (bei Codefragmenten) bzw. mit „angemessenen“ nationalen/internationalen Standards nur schwer gezeigt werden kann.

Die größere Hürde bei der Verwendung von Open Source Software stellen lizenz- und urheberrechtliche Anforderungen dar, die es kritisch zu prüfen und umzusetzen gilt.

6 Unabhängigkeit bei Functional Safety Audits und Assessments

Die ISO 26262 fordert bei der Entwicklung sicherheitsrelevanter Systeme die Durchführung verschiedener Bestätigungsmaßnahmen, insbesondere – bei Systemen mit ASIL C- und ASIL D-Einstufung – von „Functional Safety Audits“ und „Functional Safety Assessments“. Das Functional Safety Audit betrachtet hierbei die Umsetzung der geforderten Prozesse, wobei das Functional Safety Assessment die erreichte Sicherheit des entwickelten Produkts beurteilen soll. Abhängig von der Sicherheitsrelevanz des zu entwickelnden Produkts (Automotive Safety Integrity Level = ASIL) ist das Functional Safety Audit und das Functional Safety Assessment von entsprechend unabhängigen Personen durchzuführen: Je höher das vom Produkt ausgehende Risiko, umso höher die geforderte Unabhängigkeit.

Confirmation measures	Degree of independency[a] applies to ASIL				Scope
	A	B	C	D	
Functional safety audit in accordance with 6.4.8 Independence with regard to the developers of the item and project management	—	I0	I2	I3	Applies to the highest ASIL among the safety goals of the item
Functional safety assessment in accordance with 6.4.9 Independence with regard to the developers of the item and project management	—	I0	I2	I3	Applies to the highest ASIL among the safety goals of the item

[a] The notations are defined as follows:
— —: no requirement and no recommendation for or against regarding this confirmation measure;
— I0: the confirmation measure should be performed; however, if the confirmation measure is performed, it shall be performed by a different person;
— I1: the confirmation measure shall be performed, by a different person;
— I2: the confirmation measure shall be performed, by a person from a different team, i.e. not reporting to the same direct superior;
— I3: the confirmation measure shall be performed, by a person from a different department or organization, i.e. independent from the department responsible for the considered work product(s) regarding management, resources and release authority.

[b] A software tool development is outside the item's safety lifecycle whereas the qualification of such a tool is an activity of the safety lifecycle.

Abbildung 5: Geforderte Unabhängigkeit bei Functional Safety Audits und Functional Safety Assessments. Auszug aus [ISO11].

Insbesondere beim Unabhängigkeitsgrad „I3“ (Person aus einer anderen Abteilung oder Organisation) stellt sich die Frage, wie dies umgesetzt werden kann. Diesen Unabhängigkeitsgrad generell mit „firmenextern“ gleichzusetzen, ist weder von der ISO 26262 gewollt, noch notwendig, noch in jedem Fall zielführend. Hier sei insbesondere das Functional Safety Assessment genannt: Zur Beurteilung der erreichten Produktsicherheit ist sowohl Unabhängigkeit als auch in besonderem Maße Produktwissen notwendig. Fehlende Unabhängigkeit bei der Bewertung kann dazu führen, dass das Assessment nicht hinreichend objektiv durchgeführt wird. Andererseits führt fehlendes Produktwissen beim Functional Safety Assessment zu einem nutzlosen Ergebnis, da es ohne detailliertes Verständnis der im Produkt implementierten Sicherheitsmechanismen nicht möglich ist, deren Angemessenheit und Wirksamkeit zu beurteilen. Generelles Grundproblem ist, dass je unabhängiger eine Person ist, umso weniger detailliertes Produktwissen vorhanden sein wird – und umgekehrt.

Als Lösung bietet sich hier an, als verantwortlichen Functional Safety Assessor jemanden auszuwählen, der die notwendige und geforderte Unabhängigkeit mitbringt und der gegebenenfalls durch ein Expertenteam unterstützt wird, das ihn mit dem notwendigen Produktwissen unterstützt. Hierbei kann insbesondere in großen Organisationen, zum Beispiel bei einem typischen OEM oder Tier1-Zulieferer die geforderte Unabhängigkeit durchaus konzernintern dargestellt werden, wie diverse Beispiele aus der Praxis zeigen [MR11, KU12]. Insbesondere sei in diesem Zusammenhang darauf hingewiesen, dass die ISO 26262 keinerlei Zertifizierung fordert, also weder von Prozessen, noch von Produkten, Personen oder Software-Werkzeugen. Generell ist ein Zertifikat immer dahingehend zu hinterfragen, ob es das leistet, was es zu versprechen scheint – beispielsweise ob und wie in einem zugelieferten Produkt diejenigen Sicherheitsanforderungen implementiert sind, die für die funktional sichere Integration in ein übergeordnetes System notwendig sind. Ein blindes Vertrauen auf ein Zertifikat ist hier wenig sinnvoll und darf die Sorgfaltspflicht des Entwicklers bzw. Integrators nicht ersetzen.

7 Zusammenfassung und Ausblick

Die ISO 26262 wurde im November 2011 veröffentlicht und trägt aus juristischer seither zum Stand der Technik bei der Entwicklung elektrischer/elektronischer Systeme im Automobil bei. Aus technisch-inhaltlicher Sicht ist es jedoch so, dass einige Anforderungen der ISO 26262 visionär (bis hin zu illusionär) formuliert sind. Somit ist eine Umsetzung der ISO 26262 zum Veröffentlichungszeitpunkt nicht möglich, es ist eine Einführungsphase notwendig. In dieser Zeit gilt es gemeinsame Brancheninterpretationen zu finden, die der Intention der ISO 26262 möglichst nahe kommen ohne die Produktsicherheit zu gefährden. Diese stattfindende Harmonisierung bei der Entwicklung sicherer E/E-Systeme trägt maßgeblich dazu bei, dass insbesondere im ausgeprägten Netzwerk der OEM-/Zuliefererverflechtungen ein gemeinsames Verständnis entsteht, wie die ISO 26262 zu interpretieren ist und effizient umgesetzt werden kann.

Um diese Harmonisierung weiter voranzutreiben, wurde bereits die Arbeit an der 2nd Edition der ISO 26262 begonnen, mit deren Veröffentlichung ca. 2016/2017 gerechnet werden kann. In dieser ist angedacht, die übrigen Fahrzeugklassen (LKW, Busse, Motorräder) zu berücksichtigen, um auch hier ein einheitliches Branchenverständnis zu erreichen.

Eine neue Herausforderung in diesem Umfeld ergibt sich durch die stark wachsende Vernetzung von Fahrzeugen – sowohl innerhalb des Fahrzeugs selbst auch des Fahrzeugs mit seiner Umgebung (Car-2-X Kommunikation). Dies bietet vermehrt Angriffspunkte für Angriffe von außen auf das System, die – gewollt oder als Seiteneffekt – die Safety des Fahrzeugs kompromittieren können. Hier ist in Zukunft eine verstärkte Zusammenarbeit der Domänen Safety und Security notwendig [Kl12].

Literaturverzeichnis

[BMW12] BMW Gruppe: Der neue BMW C evolution. Pressemitteilung vom 27.07.2012.

[ISO11] International Organization for Standardization: ISO 26262:2011(E) – Road vehicles – Functional safety. Geneva, 2011.

[KHK11] Klauda M.; Hamann R.; Kriso S.: ISO 26262 – Was kommt da auf uns zu? In: VDI-Berichte 2132, 2011, S. 285-297.

[KH11] Kriso S.; Hamann R.: Die ISO 26262 ist veröffentlicht - Konsequenzen für OEMs und Zulieferer. In: VDI-Berichte 2132, 2011, S. 299-308.

[Kl12] Klauda, M. et al.: Automotive Safety und Security aus Sicht eines Zulieferers. In: Automotive 2012, Karlsruhe, 11/2012.

[KS09] Kriso, S.; Sauler, J.: Die Norm zur funktionalen Sicherheit ISO 26262. In: Elektronik-Praxis Sonderheft Embedded Software Engineering. Vogel-Verlag, Würzburg, 2009; S. 32-36.

[KU12] Kriso, S.; Unruh, J.: Implementation of Functional Safety Audits and Assessments at Bosch. In: IQPC-Konferenz „Experiences with ISO 26262“, München, 03/2012

[MR11] Molle, E.; Rau, M.: Bestätigungsmaßnahmen der ISO 26262 und organisatorische Umsetzungsbeispiele. In: Safetronic 2011, München, 11/2011

[Re11] Reuter, A.: Rechtliche Aspekte bei der Umsetzung der ISO 26262. Euroforum-Konferenz ISO 26262, Stuttgart, 2011.

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Lan-guage to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Business 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Er-fahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): "Heute schon das Morgen sehen"

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT:
Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting. CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik –
Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwick-lung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^{2}CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.)
10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on Electronic Voting 2010
co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2010)
Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.)
German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.)
perspeGKtive 2010
Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science – Proceedings of ISSS and BPSC

P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.)
Vom Projekt zum Produkt
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschafts-informatik (WI-MAW), Aachen, 2010

P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM`2010
Second International Workshop on Formal Methods and Agile Methods

P-180 Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, Holger Schwarz (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW)
14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)

P-181 Michael Clasen, Otto Schätzel, Brigitte Theuvsen (Hrsg.)
Qualität und Effizienz durch informationsgestützte Landwirtschaft, Fokus: Moderne Weinwirtschaft

P-182 Ronald Maier (Hrsg.)
6th Conference on Professional Knowledge Management
From Knowledge to Action

P-183 Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.)
Software Engineering 2011
Fachtagung des GI-Fachbereichs Softwaretechnik

P-184 Ralf Reussner, Alexander Pretschner, Stefan Jähnichen (Hrsg.)
Software Engineering 2011
Workshopband
(inkl. Doktorandensymposium)

P-185 Hagen Höpfner, Günther Specht, Thomas Ritz, Christian Bunse (Hrsg.)
MMS 2011: Mobile und ubiquitäre Informationssysteme Proceedings zur 6. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2011)

P-186 Gerald Eichler, Axel Küpper, Volkmar Schau, Hacène Fouchal, Herwig Unger (Eds.)
11th International Conference on Innovative Internet Community Systems (I^2CS)

P-187 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
4. DFN-Forum Kommunikations-technologien, Beiträge der Fachtagung 20. Juni bis 21. Juni 2011 Bonn

P-188 Holger Rohland, Andrea Kienle, Steffen Friedrich (Hrsg.)
DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. 5.–8. September 2011, Dresden

P-189 Thomas, Marco (Hrsg.)
Informatik in Bildung und Beruf INFOS 2011
14. GI-Fachtagung Informatik und Schule

P-190 Markus Nüttgens, Oliver Thomas, Barbara Weber (Eds.)
Enterprise Modelling and Information Systems Architectures (EMISA 2011)

P-191 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2011
International Conference of the Biometrics Special Interest Group

P-192 Hans-Ulrich Heiß, Peter Pepper, Holger Schlingloff, Jörg Schneider (Hrsg.)
INFORMATIK 2011
Informatik schafft Communities

P-193 Wolfgang Lehner, Gunther Piller (Hrsg.)
IMDM 2011

P-194 M. Clasen, G. Fröhlich, H. Bernhardt, K. Hildebrand, B. Theuvsen (Hrsg.)
Informationstechnologie für eine nachhaltige Landbewirtschaftung
Fokus Forstwirtschaft

P-195 Neeraj Suri, Michael Waidner (Hrsg.)
Sicherheit 2012
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 6. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)

P-196 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2012
Proceedings of the 11th International Conference of the Biometrics Special Interest Group

P-197 Jörn von Lucke, Christian P. Geiger, Siegfried Kaiser, Erich Schweighofer, Maria A. Wimmer (Hrsg.)
Auf dem Weg zu einer offenen, smarten und vernetzten Verwaltungskultur
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2012

P-198 Stefan Jähnichen, Axel Küpper, Sahin Albayrak (Hrsg.)
Software Engineering 2012
Fachtagung des GI-Fachbereichs Softwaretechnik

P-199 Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.)
Software Engineering 2012
Workshopband

P-200 Gero Mühl, Jan Richling, Andreas Herkersdorf (Hrsg.)
ARCS 2012 Workshops

P-201 Elmar J. Sinz Andy Schürr (Hrsg.)
Modellierung 2012

P-202 Andrea Back, Markus Bick, Martin Breunig, Key Pousttchi, Frédéric Thiesse (Hrsg.)
MMS 2012:Mobile und Ubiquitäre Informationssysteme

P-203 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
5. DFN-Forum Kommunikations-technologien
Beiträge der Fachtagung

P-204 Gerald Eichler, Leendert W. M. Wienhofen, Anders Kofod-Petersen, Herwig Unger (Eds.)
12th International Conference on Innovative Internet Community Systems (I2CS 2012)

P-205 Manuel J. Kripp, Melanie Volkamer, Rüdiger Grimm (Eds.)
5th International Conference on Electronic Voting 2012 (EVOTE2012)
Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-206 Stefanie Rinderle-Ma, Mathias Weske (Hrsg.)
EMISA 2012
Der Mensch im Zentrum der Modellierung

P-207 Jörg Desel, Jörg M. Haake, Christian Spannagel (Hrsg.)
DeLFI 2012: Die 10. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.
24.–26. September 2012

P-208 Ursula Goltz, Marcus Magnor,
Hans-Jürgen Appelrath, Herbert Matthies,
Wolf-Tilo Balke, Lars Wolf (Hrsg.)
INFORMATIK 2012

P-210 Erhard Plödereder, Peter Dencker,
Herbert Klenk, Hubert B. Keller,
Silke Spitzer (Hrsg.)
Automotive – Safety & Security 2012
Sicherheit und Zuverlässigkeit für
automobile Informationstechnik

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de